图灵程序设计丛书

MongoDB

权威指南

MongoDB: The Definitive Guide

[美] *Kristina Chodorow & Michael Dirolf* 著
Jeremy Zawodny 序
程显峰 译

O'REILLY®


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

 图灵程序设计丛书

MongoDB权威指南

MongoDB: The Definitive Guide

[美] Kristina Chodorow Michael Dirolf 著
[美] Jeremy Zawodny 序
程显峰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

MongoDB权威指南 / (美) 霍多罗夫 (Chodorow,K.),
(美) 迪洛尔夫 (Dirolf,M.) 著 ; 程显峰译. -- 北京:
人民邮电出版社, 2011.5
（图灵程序设计丛书）
书名原文: MongoDB: The Definitive Guide
ISBN 978-7-115-25112-1

Ⅰ. ①M… Ⅱ. ①霍… ②迪… ③程… Ⅲ. ①数据模型②数据库系统－程序设计 Ⅳ. ①TP311.13

中国版本图书馆CIP数据核字(2011)第048973号

内 容 提 要

本书是一本广受好评的 MongoDB 方面的图书。与传统的关系型数据库不同，MongoDB 是一种面向文档的数据库。书中介绍了面向文档的存储方式及利用 MongoDB 的无模式数据模型处理文档、集合和多个数据库，讲述了如何执行基本的写操作以及如何执行各种复杂的条件查询，还介绍了索引、聚合工具以及其他高级查询技术，另外对监控、安全性和身份验证、备份和修复、水平扩展 MongoDB 数据库等内容也有所涉及。

本书适合数据库开发人员阅读。

图灵程序设计丛书

MongoDB 权威指南

◆ 著　[美] Kristina Chodorow　Michael Dirolf
序　[美] Jeremy Zawodny
译　程显峰
责任编辑　王军花
执行编辑　李　静

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：12
字数：278千字　2011年 5月第 1 版
印数：1－4 000册　2011年 5月北京第 1 次印刷
著作权合同登记号　图字：01-2010-6929号

ISBN 978-7-115-25112-1

定价：39.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明



O’Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O’Reilly Media, Inc.授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc.是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的*The Whole Internet User’s Guide & Catalog*（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O’Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc.具有深厚的计算机专业背景，这使得O’Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media, Inc.依靠他们及时地推出图书。因为O’Reilly Media, Inc.紧密地与计算机业界联系着，所以O’Reilly Media, Inc.知道市场上真正需要什么图书。

目录

中文版序一

2010 年，NoSQL 在国内掀起了一阵热潮，其中风头最劲的莫过于 MongoDB 了。越来越多的业界公司已经或准备将 MongoDB 投入实际的生产环境，很多创业团队也将 MongoDB 作为自己的首选数据库，创造出令人眼花缭乱的移动互联网应用。

我在 2008 年开始接触 MongoDB，经历了早期的 0.8 到现在最新的 1.8，并从 0.9 将其部署到生产环境中，对 MongoDB 的迅速发展感慨良多。MongoDB 自由灵活的文档模型，让我的开发过程无比顺畅。对于大数据量、高并发、弱事务的互联网应用，MongoDB 则是一个如瑞士军刀般的利器。尽管我不认同 MongoDB 会在所有场合完全取代 MySQL，但我相信它完全可以满足 Web 2.0 和移动互联网应用的数据存储需求。MongoDB 内置的水平扩展机制提供了从百万到十亿级别的数据量处理能力，其开箱即用的特性也大大降低了中小网站的运维成本。对于创业团队，MongoDB 也是不二的选择。

从我自己的实践来看，扔掉 MySQL 的结果其实并不难，前提是你要改变思路，理解 MongoDB 的设计哲学，正视它的优缺点。做到这一点并不容易，而有手头这本书作为参考，可以让你少走很多的弯路。

在给本书写序的时候，我收到一封来自淘宝的朋友的邮件，询问我关于 Mongo shell 的一个小问题。很巧的是，问题的答案就在本书的第 2 章中。所以你看，即便是很有经验的 MongoDB 开发者，本书也能给你带来很多惊喜。

本书译者程显峰是 MongoDB 中文社区的推动者，这些年一直致力于 MongoDB 的普及和官方文档的翻译工作。我是在 2010 年他组织的国内首次社区聚会活动上认识他的，在那次聚会上，程显峰邀请 MongoDB 专家 Peter Membrey 给大家做了一次

精彩的演示，这给我留下了极为深刻的印象。

作为中译本，大家最关心的是译文是否能够原汁原味地将原书的精华完整展现出来，同时还要避免生涩歧义。初读本书，我认为译者做到了这一点。作为首本 MongoDB 方面的中文书，一些名词的翻译是难度很大的，为此，显峰也反复推敲，几经易稿，终于成文。由此可见其态度之认真。

我相信，在未来 1~2 年，MongoDB 和 Nginx、Linux、PHP/Python/Ruby 的组合，将取代原来的 LAMP 组合，让我们拭目以待。

视觉中国 CTO　潘凡

2011 年 4 月

中文版序二

初见 MongoDB 是在 2010 年年初，当时我正要启动一个互联网产品的开发工作，因此进行技术选型也是顺理成章的事情。我是个相对比较激进的技术人员，总是不甘于使用自己熟悉的东西。过去做互联网产品开发，大都使用 SQL Server、MySQL 或 PostgreSQL 等关系型数据库产品，配合现成的数据访问工具或类库，开发起来还算熟练。

但我也一直在寻找关系型数据库的替代品，因为我觉得它使用上着实有些不便。例如，为了产品中的某个实体的查询操作，我们需要把一个本属于该实体的数据拆分至另一个表中，以便进行连接查询。于是无论是创建、删除还是更新，我们要涉及的操作便增加了许多。更别说互联网项目时刻都在发展和变动，改变一个存储单元的结构是常事，至今关系型数据库的在线模式更新依旧不是件简单的事情。

总而言之，我烦。

由于没有历史包袱，我总是设法在新项目里引入一些特别的事物。这次也不例外。当时正是 NoSQL 逐渐兴起的时刻，我便去了解了一下相关的存储产品。例如 Tokyo Cabinet/Tyrant、CouchDB 或是 MongoDB 等。最终我选择了 MongoDB，因为它最为简单易用。它的集合支持松散的模式，易于灵活调整。它支持复杂的属性，并可为之建立索引，作为查询条件。它还可以直接对记录的某个字段进行原子性的改变——这在 NoSQL 产品中并不多见，例如在尝试 Tokyo Tyrant 时发现，更新一条记录需要客户端进行一次完整的读取和提交，这直接断了我用 Tokyo Tyrant 作为主要存储方式的念头。

没错，MongoDB 首先吸引我的就是它的易用性，而不是业界津津乐道的性能或是

海量数据下的表现等。因为在我看来，如今分布式环境对“单点”下的性能及海量数据支持的要求并不如想象中那么重要，Facebook 等互联网巨擘所总结来的经验，对我来说更具学习意义而不是现实意义。我想要寻找的东西，它就该是易用的。具体来说，便是对程序员友好。

我是程序员，我懒。

这么看来，可能我选择 MongoDB 有些盲目而冲动，当时除了 SourceForge 之外，似乎也没有人在用 MongoDB 作为主要存储方式。但事实上，我也花了一周多的时间观察 MongoDB 的表现。我预测了未来一年里的数据量，构造尽可能真实的测试数据和结构，在生产环境中长时间地运行以观察它的性能及稳定性表现。MongoDB 最终也没有让我们失望。我们也没有利用当时还不成熟的 Auto Sharding 功能，而是设计了能够自由水平拆分的架构方案，确保对 MongoDB 单点数据量的控制。我不能确保 MongoDB 是一个永久可靠的方案，我只是想保证：即便是以 MongoDB 目前的表现，也至少够项目发展一年。我懒得想太远。悲观地说，谁知道一年后我的项目是否还存在呢？乐观地想，到时候 MongoDB 一定发展得更好了吧！

回头看来，MongoDB 的发展势头有增无减，官方网站上的“案例”数量也有了成倍增长，可谓是如今最热门的 NoSQL 产品之一。我想这也和它的易用性不无关系，产品越是易用，则越会有人用，于是越会有更多人投入，也就越不容易失败。

当然，MongoDB 的缺点也很明显，例如它对程序员十分友好，却对系统管理员是个考验。我之前在微博上开玩笑地说：“MongoDB 的系统管理员上辈子是折翼的天使，他们牺牲自己，方便整个团队。”目前 MongoDB 在系统维护方面依旧缺少业界实践，往往只能靠系统管理员自行摸索。

看到这儿，您可能会觉得这不像是一本书的推荐或是序。但我倒觉得，其实上面这些文字正好可以用来概括本书的内容：与其说它是一本写给 MongoDB 系统管理员的书，更不如说是面向“使用 MongoDB 的程序员”。这本书细致地介绍了 MongoDB 使用的方方面面，我想作为 MongoDB 程序员的入门书籍及案头手册是十分合适的——假如您像我一样觉得官方文档缺乏条理性，不易于阅读的话。

盛大创新院研究员 赵劼
2011 年 4 月

序

10 年前，没人能预见互联网的发展给关系型数据库带来如此多的挑战。我在这个时期亲身经历了在快速发展的大型互联网公司应用 MySQL 的过程。开始时只有很少的数据，一台服务器就可以了。然后就得建立备份，以便应对大量的读取和不时的宕机。用不了多长时间，就得加一个缓存层，调整所有的查询，投入更多的硬件。

最后，你会发现自己需要将数据切分到多个集群上，并重新构建大量的应用逻辑以适应这种切分。之后不久，你又会发现被自己数月前设计的数据库结构限制住了。

怎么会呢？这是因为集群中大量的数据需要更改模式，会花费很长时间，也需要 DBA 投入相当多的宝贵时间。在代码中处理要简单一些，但也需要小型开发团队数月的努力。最后，你会不断地拷问自己有没有更好的方法，或者为什么没有在核心数据库服务器中内置更多诸如此类的功能呢。

为了应对现在 Web 应用的数据膨胀，开源社区像以往一样提供了太多的“好方法”。从内存中的键值型存储到可以使用 SQL 的 MySQL/InnoDB 变种等复杂方法，无所不有。但选择多了，做出正确的选择反而更难了。我自己就研究过其中很多种。

MongoDB 的实用性着实令人着迷。MongoDB 并不去迎合所有人的全部需求。它在功能和复杂性之间取得了很好的平衡，并且将原先十分复杂的任务大大简化。也就是说，它具备支撑今天主流 Web 应用的关键功能：索引、复制、分片、丰富的查询语法，特别灵活的数据模型。与此同时还不牺牲速度。

秉持 MongoDB 自身的风格，本书简洁明快、通俗易懂。MongoDB 新用户先看第 1 章，马上就能入门，而有经验的用户则会欣赏体验本书的广度和权威性。对于流行的客户端 API 和高级的管理主题，如复制、备份和分片，本书都是权威参考。

根据我最近每天使用 MongoDB 的经验，我相信本书会始终不离我左右的，无论安装还是进行分片或备份式集群的产品化部署，它都是我最好的助手。任何想仔细研究使用 MongoDB 的人都需要这本重要的参考书。

Craigslist 软件工程师　Jeremy Zawodny
2010 年 8 月

前言

本书的组织结构

快速起步

第 1 章将简要讲述 MongoDB 的背景：项目创立原因、希望达到的目标、选用它的理由。第 2 章会接着介绍一些 MongoDB 的核心概念和术语，还有如何上手操作数据库和 shell 的内容。

部署MongoDB

接下来的两章会介绍 MongoDB 开发者需要的基础知识。第 3 章介绍了如何执行一些基本的写入操作，包括在不同安全和速度等级下的实现细节。第 4 章主要介绍如何来查找文档和创建复杂的查询。这一章还包括如何迭代结果和其他一些用于结果处理的方法，如排序、数量限制和忽略。

进阶指南

之后的三章会深入探讨一些比存储和检索数据更复杂的用法。第 5 章将介绍索引是什么和怎么在 MongoDB 中使用，还介绍了用于检查和修改索引的工具，以及索引管理。第 6 章介绍了多种利用 MongoDB 聚集数据的方法，包括计数、查找唯一值、文档分组和 MapReduce。第 7 章会对前几章没有涉及的要点做一个补充，如文件存储、服务器端 JavaScript、数据库命令和数据库引用。

管理

接下来的三章编程的味道淡一些，侧重 MongoDB 的运行。第 8 章讨论了启动数

据库的多种选项，监控 MongoDB 服务器和维护部署的安全性。如何对存储在 MongoDB 中的数据进行合理的数据备份也在这章介绍了。第 9 章包括如何设立 MongoDB 的复制，具体包括配置标准主从集群、设置自动故障恢复。这章还会揭示复制的工作原理和调整选项。第 10 章探讨了如何水平扩展 MongoDB，包括什么是自动分片、如何设置及其对应用程序的影响。

用MongoDB开发应用

第 11 章会介绍几个使用 MongoDB 的示例应用，这些应用是使用 Java、PHP、Python 和 Ruby 编写的。这些例子展示了如何将本书前面介绍的概念应用到具体的语言和问题域中去。

附录

附录 A 介绍了 MongoDB 版本控制方案，以及如何在 Windows、OS X 和 Linux 下安装的细节。附录 B 介绍了 MongoDB shell 中一些有用的技巧和工具。附录 C 更详细地介绍了 MongoDB 的内部工作原理：存储引擎、数据格式和 MongoDB 传输协议。

本书排版规范

本书使用的排版规范如下所示。

- 楷体

用于表示新的术语。

- 等宽字体

表示程序片段，也在段落中表示程序中使用的变量、函数名、命令行实用工具、环境变量、语句和关键字等元素。

- 等宽加粗

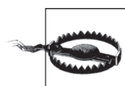
这种字体表示用户需要手动输入的命令或者相应的文本。

- 等宽斜体

用户需要根据自己所提供的值或由上下文所确定的值进行更改的部分。



这个图标代表小窍门、建议或者注意。



这个标识代表警告。

使用代码示例

让本书助你一臂之力。也许你要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

我们非常希望你能标明出处，但并不强求。出处一般含有书名、作者、出版商和 ISBN，例如"MongoDB 权威指南，Kristina Chodorow 和 Michael Dirolf（O'Reilly，2010）版权所有，978-1-449-38156-1"。

如果有关于使用代码的未尽事宜，可以随时与我们取得联系，permissions@oreilly.com。

Safari在线图书

Safari 在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500 多种技术性和创新性参考书以及视频，以便快速地找到需要的答案。

订阅后就可以访问在线图书馆内的所有页面和视频。可以在手机或其他移动设备上阅读。还能在新书上市之前抢先阅读，也能够看到还在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以节省大量时间。

这本书也在其中。欲访问本书的电子版，或者由 O'Reilly 或其他出版社出版的相关图书，请到 http://my.safaribooksonline.com 免费注册。

我们的联系方式

请把对本书的评论和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到关于本书的相关信息，包括勘误表、示例代码以及其他的信息。本书的网站地址是：

http://www.oreilly.com/catalog/9780596805784/

中文书

http://www.oreilly.com.cn/index.php?func=book&isbn=978-7-115-25112-1

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于本书的更多信息、会议、资源中心和网络，请访问以下网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

感谢 Eliot Horowitz 和 Dwight Merriman，是他们启动了 MongoDB 项目，才使得这一切成为可能。感谢技术审稿人 Alberto Lerner、Mathias Stearn、Aaron Staple、James Avery 和 John Hornbeck，他们在完善本书的过程中做出了不懈努力。感谢优秀的编辑 Julie Steele 在本书出版的每一阶段对我们的帮助。还要感谢 O’ Reilly 对本书出版做出贡献的其他工作人员。最后，非常感谢 MongoDB 社区从始至终对这个项目和本书的支持。

Kristina感谢以下人员

感谢我在 10gen 的所有同事，他们与我分享了 MongoDB 的知识和建议（还有那些关于运营、啤酒、飞机失事的见解）。还要感谢 Mike，他像变魔术一样完成了本书的一半工作，还修正了我的很多低级错误，让它们没有机会被 Julie 看到。最后要谢谢 Andrew、Susan 和 Andy，感谢他们的支持、耐心和建议。没有你们就没有这本书。

Michael感谢以下人员

谢谢我所有的朋友，他们在本书的写作期间给了我很多鼓励。谢谢我在 10gen 的各位同事，是他们的努力才让 MongoDB 如此成功。感谢 Kristina，和她合作很愉快。最应该感谢我的家人，他们给了我无尽的支持。

第 1 章 简介

MongoDB 是一种强大、灵活、可扩展的数据存储方式。它扩展了关系型数据库的众多有用功能，如辅助索引、范围查询（range query）和排序。MongoDB 的功能非常丰富，比如内置的对 MapReduce 式聚合的支持，以及对地理空间索引的支持。

要是不能用的话，再牛的技术也是空谈，MongoDB 致力于容易上手、便于使用。MongoDB 的数据模型对开发者来说非常友好，配置选项对于管理员来说也很轻松，并且有驱动程序和数据库 shell 提供的自然语言式的 API。MongoDB 会为你扫除障碍，让你关注编程本身而不是为存储数据烦恼。

1.1 丰富的数据模型

MongoDB 是面向文档的数据库，不是关系型数据库。放弃关系模型的主要原因就是为了获得更加方便的扩展性，当然还有其他好处。

基本的思路就是将原来“行”（row）的概念换成更加灵活的“文档”（document）模型。面向文档的方式可以将文档或者数组内嵌进来，所以用一条记录就可以表示非常复杂的层次关系。使用面向对象语言的开发者恰恰这么看待数据，所以感觉非常自然。

MongoDB 没有模式：文档的键不会事先定义也不会固定不变。由于没有模式需要更改，通常不需要迁移大量数据。不必将所有数据都放到一个模子里面，应用层可以处理新增或者丢失的键。这样开发者可以非常容易地变更数据模型。

1.2 容易扩展

应用数据集的大小在飞速增加。传感器技术的发展、带宽的增加，以及可连接到因

特网的手持设备的普及使得当下即便很小的应用也要存储大量数据，量大到很多数据库都应付不来。T 级别的数据原来是闻所未闻的，现在已经司空见惯了。

由于开发者要存储的数据不断增长，他们面临一个非常困难的选择：该如何扩展他们的数据库？升级呢（买台更好的机器），还是扩展呢（将数据分散到很多机器上）？升级通常是最省力气的做法，但是问题也显而易见：大型机一般都非常昂贵，最后达到了物理极限的话花多少钱都买不到更好的机器。对于大多数人希望构建的大型 Web 应用来说，这样做既不现实也不划算。而扩展就不同了，不但经济而且还能持续添加：想要增加存储空间或者提升性能，只需要买台一般的服务器加入集群就好了。

MongoDB 从最初设计的时候就考虑到了扩展的问题。它所采用的面向文档的数据模型使其可以自动在多台服务器之间分割数据。它还可以平衡集群的数据和负载，自动重排文档。这样开发者就可以专注于编写应用，而不是考虑如何扩展。要是需要更大的容量，只需在集群中添加新机器，然后让数据库来处理剩下的事。

1.3 丰富的功能

很难界定什么才算作一个功能：上面提及的算是功能吗？关系型数据库做不到的算吗？ Memcached 做不到的呢？其他面向文档的数据库做不到的又如何呢？但无论界定的标准是什么，都可以说 MongoDB 拥有一些真正独特的、好用的工具，其他方案不具备或不完全具备这些工具。

- 索引
  MongoDB 支持通用辅助索引，能进行多种快速查询，也提供唯一的、复合的和地理空间索引能力。

- 存储 JavaScript
  开发人员不必使用存储过程了，可以直接在服务端存取 JavaScript 的函数和值。

- 聚合
  MongoDB 支持 MapReduce 和其他聚合工具。

- 固定集合
  集合的大小是有上限的，这对某些类型的数据（比如日志）特别有用。

- 文件存储
  MongoDB 支持用一种容易使用的协议存储大型文件和文件的元数据。

有些关系型数据库的常见功能 MongoDB 并不具备，比如联接（join）和复杂的多

行事务。这个架构上的考虑是为了提高扩展性，因为这两个功能实在很难在一个分布式系统上实现。

1.4 不牺牲速度

卓越的性能是 MongoDB 的主要目标，也极大地影响了设计上的很多决定。MongoDB 使用 MongoDB 传输协议作为与服务器交互的主要方式（与之对应的协议需要更多的开销，如 HTTP/REST）。它对文档进行动态填充，预分配数据文件，用空间换取性能的稳定。默认的存储引擎中使用了内存映射文件，将内存管理工作交给操作系统去处理。动态查询优化器会“记住”执行查询最高效的方式。总之，MongoDB 在各个方面都充分考虑了性能。

虽然 MongoDB 功能强大，尽量保持关系型数据库的众多特性，但是它并不是要具备所有的关系型数据库的功能。它尽可能地将服务器端处理逻辑交给客户端（由驱动程序或者用户的应用程序处理）。这样精简的设计使得 MongoDB 获得了非常好的性能。

1.5 简便的管理

MongoDB 尽量让服务器自治来简化数据库的管理。除了启动数据库服务器之外，几乎没有什么必要的管理操作。如果主服务器挂掉了，MongoDB 会自动切换到备份服务器上，并且将备份服务器提升为活跃服务器。在分布式环境下，集群只需要知道有新增加的节点，就会自动集成和配置新节点。

MongoDB 的管理理念就是尽可能地让服务器自动配置，让用户能在需要的时候调整设置（但不强制）。

1.6 其他内容

在本书中，我们还会花些时间追溯一下在开发 MongoDB 的过程中一些决定的原因和动机，希望通过这种方式来阐释 MongoDB 的理念。毕竟，MongoDB 的愿景是对自身最好的诠释——建立一种灵活、高效、易于扩展、功能完备的数据库。

第 2 章 入门

MongoDB 非常强大，同时也很容易上手。本章会介绍一些 MongoDB 的基本概念。

- *文档*是 MongoDB 中数据的基本单元，非常类似于关系数据库管理系统中的行（但是比行要复杂得多）。
- 类似地，*集合*可以被看做是没有模式的表。
- MongoDB 的单个实例可以容纳多个独立的*数据库*，每一个都有自己的集合和权限。
- MongoDB 自带简洁但功能强大的 JavaScript shell，这个工具对于管理 MongoDB 实例和操作数据作用非常大。
- 每一个文档都有一个特殊的键 `"_id"`，它在文档所处的集合中是唯一的。

2.1 文档

*文档*是 MongoDB 的核心概念。多个键及其关联的值有序地放置在一起便是文档。每种编程语言表示文档的方法不太一样，但大多数编程语言都有相通的一种数据结构，比如映射、散列或字典。例如，在 JavaScript 里面，文档表示为对象：

```
{"greeting" : "Hello, world!"}
```

这个文档只有一个键 `"greeting"`，其对应的值为 `"Hello, world!"`。绝大多数情况下，文档会比这个简单的例子复杂得多，经常会包含多个键 / 值对：

```
{"greeting" : "Hello, world!", "foo" : 3}
```

这个例子很好地解释了几个十分重要的概念。

- 文档中的键 / 值对是有序的，上面的文档和下面的文档是完全不同的：

```
{"foo" : 3, "greeting" : "Hello, world!"}
```



通常文档中键的顺序并不重要。实际上，有些编程语言默认对文档的呈现根本就不顾忌顺序（如 Python 的字典，Perl 和 Ruby 1.8 中的散列）。这些语言的驱动包含特殊的机制，会在少数必要的情况下指定文档的排序。本书会时常提到这些情况。

- 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档,详见 2.6.5 节）。这个例子中 "greeting" 的值是个字符串，而 "foo" 的值是个整数。

文档的键是字符串。除了少数例外情况，键可以使用任意 UTF-8 字符。

- 键不能含有 \0（空字符）。这个字符用来表示键的结尾。
- . 和 $ 有特别的意义，只有在特定环境下才能使用，后面的章节会详细说明。通常来说就是被保留了，使用不当的话，驱动程序会提示。
- 以下划线“_”开头的键是保留的，虽然这个并不是严格要求的。

MongoDB 不但区分类型，也区分大小写。例如，下面的两个文档是不同的：

```
{"foo" : 3}
{"foo" : "3"}
```

以下的文档也是不同的：

```
{"foo" : 3}
{"Foo" : 3}
```

还有一个非常重要的事项需要注意，MongoDB 的文档不能有重复的键。例如，下面的文档是非法的：

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

2.2 集合

集合就是一组文档。如果说 MongoDB 中的文档类似于关系型数据库中的行，那么集合就如同表。

2.2.1 无模式

集合是无模式的。这意味着一个集合里面的文档可以是各式各样的。例如，下面两个文档可以存在于同一个集合里面：

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

注意，上面的文档不光是值的类型不同（字符串和整数），它们的键也是完全不一样的。因为集合里面可以放置任何文档，随之而来的一个问题是："还有必要使用多个集合吗？"问得好！要是没必要对各种文档划分模式，那么为什么还要使用多个集合呢？下面是一些理由。

- 把各种各样的文档都混在一个集合里面，无论对于开发者还是管理员来说都是噩梦。开发者要么确保每次查询只返回需要的文档种类，要么让执行查询的应用程序来处理所有不同类型的文档。如果查询博客文章还要剔除那些含有作者数据的文档，就很令人恼火。
- 在一个集合里面查询特定类型的文档在速度上也很不划算，分开做多个集合要快得多。例如，集合里面有个标注类型的键，现在查询其值为"skim"、"whole"或"chunky monkey"的文档，就会非常慢。如果按照名字分割成3个集合的话，查询会快很多（参见"子集合"部分）
- 把同种类型的文档放在一个集合里，这样数据会更加集中。从只含有博客文章的集合里面查询几篇文章，会比从含有文章和作者数据的集合里面查出几篇文章少消耗磁盘寻道操作。
- 当创建索引的时候，文档会有附加的结构（尤其是有唯一索引的时候）。索引是按照集合来定义的。把同种类型的文档放入同一个集合里面，可以使索引更加有效。

你已经看到了，的确有很多理由创建一个模式把相关类型的文档规整到一起。但MongoDB对此还是不做强制要求，让开发者更有灵活性。

2.2.2 命名

我们可以通过名字来标识集合。集合名可以是满足下列条件的任意UTF-8字符串。

- 集合名不能是空字符串""。
- 集合名不能含有\0字符（空字符），这个字符表示集合名的结尾。
- 集合名不能以"system."开头，这是为系统集合保留的前缀。例如system.users这个集合保存着数据库的用户信息，system.namespaces集合保存着所有数据库集合的信息。
- 用户创建的集合名字不能含有保留字符$。有些驱动程序的确支持在集合名里面包含$，这是因为某些系统生成的集合中包含该字符。除非你要访问这种系统创建的集合，否则千万不要在名字里出现$。

子集合

组织集合的一种惯例是使用"."字符分开的按命名空间划分的子集合。例如，一个

带有博客功能的应用可能包含两个集合，分别是 `blog.posts` 和 `blog.authors`。这样做的目的只是为了使组织结构更好些，也就是说 `blog` 这个集合（这里根本就不需要存在）及其子集合没有任何关系。

虽然子集合没有特别的地方，但还是很有用，很多 MongoDB 工具中都包含子集合。

- GridFS 是一种存储大文件的协议，使用子集合来存储文件的元数据，这样就与内容块分开了（关于 GridFS 详见第 7 章）。
- MongoDB 的 Web 控制台通过子集合的方式将数据组织在 DBTOP 部分（关于管理详见第 8 章）。
- 绝大多数驱动程序都提供语法糖，为访问指定集合的子集合提供方便。例如，在数据库 shell 里面，`db.blog` 代表 `blog` 集合，`db.blog.posts` 代表 `blog.posts` 集合。

在 MongoDB 中使用子集合来组织数据是很好的方法，在此强烈推荐。

2.3 数据库

MongoDB 中多个文档组成集合，同样多个集合可以组成*数据库*。一个 MongoDB 实例可以承载多个数据库，它们之间可视为完全独立的。每个数据库都有独立的权限控制，即便是在磁盘上，不同的数据库也放置在不同的文件中。将一个应用的所有数据都存储在同一个数据库中的做法就很好。要想在同一个 MongoDB 服务器上存放多个应用或者用户的数据，就要使用不同的数据库了。

和集合一样，数据库也通过名字来标识。数据库名可以是满足以下条件的任意 UTF-8 字符串。

- 不能是空字符串（""）。
- 不得含有 ' '（空格）、. 、$ 、/、\ 和 \0（空字符）。
- 应全部小写。
- 最多 64 字节。

要记住一点，数据库名最终会变成文件系统里的文件，这也就是有如此多限制的原因。

有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库。这些数据库如下所示。

- admin

  从权限的角度来看，这是“root”数据库。要是将一个用户添加到这个数据库，

这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。

- local

这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合（关于复制和本地数据库详见第 9 章）

- config

当 Mongo 用于分片设置时（参见第 10 章），config 数据库在内部使用，用于保存分片的相关信息。

把数据库的名字放到集合名前面，得到的就是集合的完全限定名，称为命名空间。例如，如果你在 cms 数据库中使用 `blog.posts` 集合，那么这个集合的命名空间就是 `cms.blog.posts`。命名空间的长度不得超过 121 字节，在实际使用当中应该小于 100 字节。关于 MongoDB 中集合的命名空间和内部表示的更多信息，可以参考附录 C。

2.4 启动MongoDB

MongoDB 一般作为网络服务器来运行，客户端可以连接到该服务器并执行操作。要启动该服务器，需要运行 `mongod` 可执行文件：

```
$ ./mongod
./mongod --help for help and startup options
Sun Mar 28 12:31:20 Mongo DB : starting : pid = 44978 port = 27017
dbpath = /data/db/ master = 0 slave = 0 64-bit
Sun Mar 28 12:31:20 db version v1.5.0-pre-, pdfile version 4.5
Sun Mar 28 12:31:20 git version: ...
Sun Mar 28 12:31:20 sys info: ...
Sun Mar 28 12:31:20 waiting for connections on port 27017
Sun Mar 28 12:31:20 web admin interface listening on port 28017
```

或者在 Windows 下，这样操作：

```
$ mongod.exe
```



关于安装 MongoDB 的详细信息，参见附录 A。

`mongod` 在没有参数的情况下会使用默认数据目录 /data/db（在 Windows 下是 C:\data\db\)，并使用 27017 端口。如果数据目录不存在或者不可写，服务器会启动失败。所以在启动 MongoDB 前，创建数据目录（比如 mkdir -p /data/db)，并确保

对该目录有写权限是很重要的。如果端口被占用，启动也会失败。通常这是由于MongoDB 实例已经在运行了。

服务器会打印版本和系统信息，然后等待连接。默认情况下，MongoDB 监听 27017 端口。

mongod 还会启动一个非常基本的 HTTP 服务器，监听数字比主端口号高 1000 的端口，也就是 28017 端口。这意味着你可以通过浏览器访问 http://localhost:28017 来获取数据库的管理信息。

在启动服务器的 shell 下可以键入 Ctrl-C 来完全地停止 mongod 的运行。



想要了解启动和停止 MongoDB 的更多细节，请参见 8.1 节；想要了解管理接口的更多内容，可以参见 8.2.1 节。

2.5 MongoDB shell

MongoDB 自带一个 JavaScript shell，可以从命令行与 MongoDB 实例交互。这个 shell 非常有用，通过它可以执行管理操作、检查运行实例，亦或做其他尝试。这个 mongo shell 对于使用 MongoDB 来说是至关重要的工具，本书后面也会经常使用这个工具。

2.5.1 运行shell

运行 mongo 启动 shell：

```
$ ./mongo
MongoDB shell version: 1.6.0
url: test
connecting to: test
type "help" for help
>
```

shell 会在启动时自动连接 MongoDB 服务器，所以要确保在使用 shell 之前启动 mongod。

shell 是功能完备的 JavaScript 解释器，可以运行任何 JavaScript 程序。为了证明这一点，我们运行几个简单的数学运算：

```
> x = 200
200
> x / 5;
40
```

还可以充分利用 JavaScript 的标准库。

```
> Math.sin(Math.PI / 2);
1
> new Date("2010/1/1");
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

也可以定义和调用 JavaScript 函数：

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

注意，可以使用多行命令。这个 shell 会检测输入的 JavaScript 语句是否写完，如没写完还可以在下一行接着写。

2.5.2 MongoDB客户端

虽然能运行任意 JavaScript 程序很酷，但 shell 的真正威力还在于它是一个独立的 MongoDB 客户端。开启的时候，shell 会连到 MongoDB 服务器的 test 数据库，并将这个数据库连接赋值给全局变量 `db`。这个变量是通过 shell 访问 MongoDB 的主要入口点。

shell 还有些非 JavaScript 语法的扩展，是为了方便习惯于 SQL shell 的用户而添加的。这些扩展并不提供额外的功能，但它们是很棒的语法糖。例如，最重要的操作之一就是选择要使用的数据库：

```
> use foobar
switched to db foobar
```

现在如果看看 db，会发现其指向 `foobar` 数据库：

```
> db
foobar
```

因为这是一个 JavaScript shell，所以键入一个变量会将变量的值转换为字符串（这里就是数据库名）并打印出来。

可以通过 `db` 变量来访问其中的集合。例如 `db.baz` 返回当前数据库的 `baz` 集合。既然现在可以在 shell 中访问集合，那么基本上就可以执行几乎所有的数据库操作了。

2.5.3 shell中的基本操作

在 shell 查看操作数据会用到 4 个基本操作：创建、读取、更新和删除 (CRUD)。

1. 创建

insert 函数添加一个文档到集合里面。例如，假设要存储一篇博客文章。首先，创建一个局部变量 post，内容是代表文档的 JavaScript 对象。里面会有 "title"，"content" 和 "date"（发表日期）几个键。

```
> post = {"title" : "My Blog Post",
... "content" : "Here's my blog post.",
... "date" : new Date()}
{
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

这个对象是个有效的 MongoDB 文档，所以可以用 insert 方法将其保存到 blog 集合中：

```
> db.blog.insert(post)
```

这篇文章已经被存到数据库里面了。可以调用集合的 find 方法来查看一下：

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

除了我们输入的键 / 值对都完整地被保存下来，还有一个额外添加的键 "_id"。本章的最后会解释 "_id" 突然出现的原因。

2. 读取

find 会返回集合里面所有的文档。若只是想查看一个文档，可以用 findOne：

```
> db.blog.findOne()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

find 和 findOne 可以接受查询文档形式的限定条件。这将通过查询限制匹配的文档。使用 find 时，shell 自动显示最多 20 个匹配的文档，但可以获取更多文档。关

于查询的更多内容，参见第 4 章。

3. 更新

如果要更改博客文章，就要用到 update 了。update 接受（至少）两个参数：第一个是要更新文档的限定条件，第二个是新的文档。假设决定给我们先前写的文章增加评论内容，则需要增加一个新的键，对应的值是存放评论的数组。

第一步修改变量 post，增加 "comments" 键：

```
> post.comments = []
[ ]
```

然后执行 update 操作，用新版本的文档替换标题为“My Blog Post”的文章：

```
> db.blog.update({title : "My Blog Post"}, post)
```

文档已经有了 "comments" 键。再用 find 查看一下，可以看到新的键：

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
    "comments" : [ ]
}
```

4. 删除

remove 用来从数据库中永久性地删除文档。在不使用参数进行调用的情况下，它会删除一个集合内的所有文档。它也可以接受一个文档以指定限定条件。例如，下面的命令会删除我们刚刚创建的文章：

```
> db.blog.remove({title : "My Blog Post"})
```

集合现在又是空的了。

2.5.4 使用shell的窍门

由于 mongo 是个 JavaScript shell，通过在线查看 JavaScript 的文档能获得很多帮助。shell 本身内置了帮助文档，可以通过 help 命令查看。

```
> help
HELP
    show dbs                    show database names
```

```
show collections        show collections in current database
show users              show users in current database
show profile            show recent system.profile entries w. time >= 1ms
use <db name>           set current database to <db name>
db.help()               help on DB methods
db.foo.help()           help on collection methods
db.foo.find()           list objects in collection foo
db.foo.find( { a : 1 } )  list objects in foo where a == 1
it                      result of the last line evaluated
```

使用 `db.help()` 可以查看数据库级别的命令的帮助，集合的相关帮助可以通过 `db.foo.help()` 来查看。

有个了解函数功用的技巧，就是在输入的时候不要输括号。这样就会显示该函数的 JavaScript 源代码。例如，如果想看看 `update` 的机理，或者就是为了看看参数顺序，可以这么做：

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
                       upsert ? true : false, multi ? true : false);
    }
```

要查看 shell 提供的所有自动生成的 JavaScript 函数 API 文档，可访问 http://api.mongodb.org/js。

蹩脚的集合名

使用 `db.` 集合名的方式来访问集合一般不会有问题，但如果集合名恰好是数据库类的一个属性就有问题了。例如，要访问 `version` 这个集合，使用 `db.version` 就不行，因为 `db.version` 是个数据库函数（它返回正在运行的 MongoDB 服务器的版本）。

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
```

只有当 JavaScript 在 `db` 中找不到指定的属性时，才会将其作为集合返回。当有属性与目标集合同名时，可以使用 `getCollection` 函数：

```
> db.getCollection("version");
test.version
```

要查看名称中含有无效 JavaScript 字符的集合，这个函数也可以派上用场。比如 `foo-`

bar 是个有效的集合名，但是在 JavaScript 中就变成了变量相减了。通过 db.getCollection("foo-bax") 可以得到 foo-bax 集合。

在 JavaScript 中，x.y 与 x['y'] 完全等价。这就意味着不但可以直“呼”其名，也可以使用变量来访问子集合。也就是说，当需要对 blog 的每个子集合执行操作时，只需要像下面这样迭代就好了：

```
var collections = ["posts", "comments", "authors"];

for (i in collections) {
    doStuff(db.blog[collections[i]]);
}
```

而不是使用下面这种笨笨的写法：

```
doStuff(db.blog.posts);
doStuff(db.blog.comments);
doStuff(db.blog.authors);
```

2.6 数据类型

本章的开始讲了一些文档的基本概念，现在大家应该会启动并运行 MongoDB，在 shell 里面动手试一试。这一部分会更加深入一些。MongoDB 支持将多种数据类型作为文档中的值。本节我们就来逐一看看。

2.6.1 基本数据类型

MongoDB 的文档类似于 JSON，在概念上和 JavaScript 中的对象神似。JSON 是一种简单的表示数据的方式，其规范可用一段文字描述（请到 http://www.json.org 自行验证），仅包含 6 种数据类型。这带来很多好处：易于理解、易于解析、易于记忆。但另外一方面，JSON 的表现力也有限制，因为只有 null、布尔、数字、字符串、数组和对象几种类型。

虽然这些类型的表现力已经足够强大，但是对于绝大多数应用来说还需要另外一些不可或缺的类型，尤其是与数据库打交道的那些应用。例如，JSON 没有日期类型，这会使得处理本来简单的日期问题变得非常繁琐。只有一种数字类型，没法区分浮点数和整数，更不能区分 32 位和 64 位数字。也没有办法表示其他常用类型，如正则表达式或函数。

MongoDB 在保留 JSON 基本的键 / 值对特性的基础上，添加了其他一些数据类型。在不同的编程语言下这些类型的表示有些许差异，下面列出了 MongoDB 通常支持的一些类型，同时说明了在 shell 中这些类型是如何表示为文档的一部分的。

- `null`

`null` 用于表示空值或者不存在的字段。

```
{"x" : null}
```

- 布尔

布尔类型有两个值 `'true'` 和 `'false'`：

```
{"x" : true}
```

- 32 位整数

shell 中这个类型不可用。前面提到，JavaScript 仅支持 64 位浮点数，所以 32 位整数会被自动转换。

- 64 位整数

shell 也不支持这个类型。shell 会使用一个特殊的内嵌文档来显示 64 位整数，详见 2.6.2 节。

- 64 位浮点数

shell 中的数字都是这种类型。下面是一个浮点数：

```
{"x" : 3.14}
```

这个也是浮点数：

```
{"x" : 3}
```

- 字符串

UTF-8 字符串都可表示为字符串类型的数据：

```
{"x" : "foobar"}
```

- 符号

shell 不支持这种类型。shell 将数据库里的符号类型转换成字符串。

- 对象 id

对象 id 是文档的 12 字节的唯一 ID。详见 2.6.6 节：

```
{"x" : ObjectId()}
```

- 日期

日期类型存储的是从标准纪元开始的毫秒数。不存储时区：

```
{"x" : new Date()}
```

- 正则表达式

文档中可以包含正则表达式，采用 JavaScript 的正则表达式语法：

```
{"x" : /foobar/i}
```

- 代码

文档中还可以包含 JavaScript 代码：

```
{"x" : function() { /* ... */ }}
```

- 二进制数据

二进制数据可以由任意字节的串组成。不过 shell 中无法使用。

- 最大值

BSON 包括一个特殊类型，表示可能的最大值。shell 中没有这个类型。

- 最小值

BSON 包括一个特殊类型，表示可能的最小值。shell 中没有这个类型。

- 未定义

文档中也可以使用未定义类型（JavaScript 中 `null` 和 `undefined` 是不同的类型）。

```
{"x" : undefined}
```

- 数组

值的集合或者列表可以表示成数组：

```
{"x" : ["a", "b", "c"]}
```

- 内嵌文档

文档可以包含别的文档，也可以作为值嵌入到父文档中：

```
{"x" : {"foo" : "bar"}}
```

2.6.2 数字

JavaScript 中只有一种"数字"类型。因为 MongoDB 中有 3 种数字类型（32 位整数、64 位整数和 64 位浮点数），shell 必须绕过 JavaScript 的限制。默认情况下，shell 中的数字都被 MongoDB 当做是双精度数。这意味着如果你从数据库中获得的是一个 32 位整数，修改文档后，将文档存回数据库的时候，这个整数也被转换成了浮点数，即便保持这个整数原封不动也会这样的。所以明智的做法是尽量不要在 shell 下覆盖整个文档。（关于修改指定键的值的内容参见第 3 章。）

数字只能表示为双精度数（64 位浮点数）的另外一个问题是，有些 64 位的整数并不能

精确地表示为 64 位浮点数。所以，要是存入了一个 64 位整数，然后在 shell 中查看，它会显示一个内嵌文档，表示可能不准确。例如，保存一个文档[1]，其中 "myInteger" 键的值设为一个 64 位整数——3，然后在 shell 中查看，应该是下面这样的：

```
> doc = db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa08"),
    "myInteger" : {
        "floatApprox" : 3
    }
}
```

数据库中的数字是不会改变的（除非你修改了，尔后又通过 shell 保存回去了，这样它就会被转换成浮点类型）；内嵌文档只表示 shell 显示的是一个用 64 位浮点数近似表示的 64 位整数。若是内嵌文档只有一个键的话，实际上这个值是准确的。

要是插入的 64 位整数不能精确地作为双精度数显示，shell 会添加两个键，"top" 和 "bottom"，分别表示高 32 位和低 32 位。例如，如果插入 9 223 372 036 854 775 807，shell 会这样显示：

```
> db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa09"),
    "myInteger" : {
        "floatApprox" : 9223372036854776000,
        "top" : 2147483647,
        "bottom" : 4294967295
    }
}
```

"floatApprox" 是一种特殊的内嵌文档，可以作为值和文档来操作：

```
> doc.myInteger + 1
4

> doc.myInteger.floatApprox
3
```

32 位的整数都能用 64 位的浮点数精确表示，所以显示起来没什么特别的。

2.6.3 日期

在 JavaScript 中，Date 对象用做 MongoDB 的日期类型，创建一个新的 Date 对象时，通常会调用 new Date(…) 而不只是 Date(…)。调用构造函数（也就是说不包括 new）实际上会返回对日期的字符串表示，而不是真正的 Date 对象。这不是 MongoDB 的特性，而是 JavaScript 本身的特性。要是不小心忘了使用 Date 构造函

译注1：显然不是在shell中保存的。

数，最后就会导致日期和字符串混淆。字符串和日期不能互相匹配，所以这会给删除、更新、查询等很多操作带来问题。

关于 JavaScript 中 `Date` 类的详细说明和可接受的构造函数的格式，请参见 ECMA-Script 规范 15.9 节（可在 http://www.ecmascript.org 下载）。

shell 中的日期显示时使用本地时区设置。但是，日期在数据中是以从标准纪元开始的毫秒数的形式存储的，没有与之相关的时区信息（当然可以把时区信息作为其他键的值存储）。

2.6.4　数组

数组是一组值，既可以作为有序对象（像列表、栈或队列）来操作，也可以作为无序对象（像集合）操作。

在下面的文档中，`"things"` 这个键的值就是一个数组：

```
{"things" : ["pie", 3.14]}
```

从这个例子可以看到，数组可以包含不同数据类型的元素（这个例子中是一个字符串和一个浮点数）。实际上，常规键 / 值对支持的值都可以作为数组的元素，甚至是套嵌数组。

文档中的数组有个奇妙的特性，就是 MongoDB 能“理解”其结构，并知道如何“深入”数组内部对其内容进行操作。这样就能用内容对数组进行查询和构建索引了。例如，之前的例子中，MongoDB 可以查询所有 `"things"` 数组中含有 3.14 的文档。要是经常使用这个查询，可以对 `"things"` 创建索引，来提高性能。

MongoDB 可以使用原子更新修改数组中的内容，比如深入数组内部将 pie 改为 pi。在本书后面还会介绍更多这种操作的例子。

2.6.5　内嵌文档

内嵌文档就是把整个 MongoDB 文档作为另一个文档中键的一个值。这样数据可以组织得更自然些，不用非得存成扁平结构的。

例如，用一个文档来表示一个人，同时还要保存他的地址，可以将地址内嵌到 `"address"` 文档中：

```
{
    "name" : "John Doe",
    "address" : {
```

```
            "street" : "123 Park Street",
            "city" : "Anytown",
            "state" : "NY"
        }
    }
```

上面例子中 "address" 的值是另一个文档，这个文档有自己的 "street"、"city" 和 "state" 键值。

同数组一样，MongoDB 能够“理解”内嵌文档的结构，并能“深入”其中构建索引、执行查询，或者更新。

我们会在后面深入讨论模式设计，但就算是从这个简单的例子也可以看出内嵌文档可以改变处理数据的方式。在关系型数据库中，之前的文档一般会被拆分成两个表（“people”和“address”）中的两个行。在 MongoDB 中，就可以将地址文档直接嵌入人员文档中。使用得当的话，内嵌文档会使信息表示得更加自然（通常也会更高效）。

这样做也有坏处，因为 MongoDB 会储存更多重复的数据，这样是反规范化的。如果在关系数据库中“address”在一个独立的表中，要修复地址中的拼写错误。当我们对“people”和“address”执行连接操作时，每一个使用这个地址的人的信息都会得到更新。但是在 MongoDB 中，则需要在每个人的文档中修正拼写错误。

2.6.6 _id和ObjectId

MongoDB 中存储的文档必须有一个 "_id" 键。这个键的值可以是任何类型的，默认是个 ObjectId 对象。在一个集合里面，每个文档都有唯一的 "_id" 值，来确保集合里面每个文档都能被唯一标识。如果有两个集合的话，两个集合可以都有一个值为 123 的 "_id" 键，但是每个集合里面只能有一个 "_id" 是 123 的文档。

1. ObjectId

ObjectId 是 "_id" 的默认类型。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。这是 MongoDB 采用 ObjectId，而不是其他比较常规的做法（比如自动增加的主键）的主要原因，因为在多个服务器上同步自动增加主键值既费力还费时。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。后面会看到 ObjectId 类型在分片环境中要容易生成得多。

ObjectId 使用 12 字节的存储空间，每个字节两位十六进制数字，是一个 24 位的字符串。由于看起来很长，不少人会觉得难以处理。但关键是要知道这个长长的 ObjectId 是实际存储数据的两倍长。

如果快速连续创建多个 ObjectId，会发现每次只有最后几位数字有变化。另外，

中间的几位数字也会变化（要是在创建的过程中停顿几秒钟）。这是 ObjectId 的创建方式导致的。12 字节按照如下方式生成：

```
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11
时间戳         | 机器       | PID  | 计数器
```

前 4 个字节是从标准纪元开始的时间戳，单位为秒。这会带来一些有用的属性。

- 时间戳，与随后的 5 个字节组合起来，提供了秒级别的唯一性。
- 由于时间戳在前，这意味着 ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率，但是这个是没有保证的，仅仅是“大致”。
- 这 4 个字节也隐含了文档创建的时间。绝大多数驱动都会公开一个方法从 ObjectId 获取这个信息。

因为使用的是当前时间，很多用户担心要对服务器进行时间同步。其实没有这个必要，因为时间戳的实际值并不重要，只要其总是不停增加就好了（每秒一次）。

接下来的 3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的 ObjectId，不产生冲突。

为了确保在同一台机器上并发的多个进程产生的 ObjectId 是唯一的，接下来的两字节来自产生 ObjectId 的进程标识符（PID）。

前 9 字节保证了同一秒钟不同机器不同进程产生的 ObjectId 是唯一的。后 3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的 ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有 256^3（16 777 216）个不同的 ObjectId。

2. 自动生成_id

前面讲到，如果插入文档的时候没有 "_id" 键，系统会自动帮你创建一个。可以由 MongoDB 服务器来做这件事情，但通常会在客户端由驱动程序完成。理由如下。

- 虽然 ObjectId 设计成轻量型的，易于生成，但是毕竟生成的时候还是产生开销。在客户端生成体现了 MongoDB 的设计理念：能从服务器端转移到驱动程序来做的事，就尽量转移。这种理念背后的原因是，即便是像 MongoDB 这样的可扩展数据库，扩展应用层也要比扩展数据库层容易得多。将事务交由客户端来处理，就减轻了数据库扩展的负担。

- 在客户端生成 ObjectId，驱动程序能够提供更加丰富的 API。例如，驱动程序可以有自己的 insert 方法，可以返回生成的 ObjectId，也可以直接将其插入文档。如果驱动程序允许服务器生成 ObjectId，那么将需要单独的查询，以确定插入的文档中的 "_id" 值。

第 3 章

创建、更新及删除文档

本章会介绍基本的数据库移入 / 移出数据的操作，具体包含如下操作。

- 向集合添加新文档
- 从集合里删除文档
- 更新现有文档
- 为这些操作选择合适的安全级别和速度

3.1 插入并保存文档

插入是向 MongoDB 中添加数据的基本方法。对目标集使用 insert 方法，插入一个文档：

```
> db.foo.insert({"bar" : "baz"})
```

这个操作会给文档增加一个 "_id" 键（要是原来没有的话），然后将其保存到 MongoDB 中。

3.1.1 批量插入

如果要插入多个文档，使用批量插入会快一些。批量插入能传递一个由文档构成的数组给数据库。

一次发送数十、数百乃至数千个文档会明显提高插入的速度。一次批量插入只是单个的 TCP 请求，也就是说避免了许多零碎的请求所带来的开销。由于无需处理大量的消息头，这样能够减少插入时间。要知道当单个文档发送至数据库时，会有一个头部信息，告诉数据库对指定的集合做插入操作。用批量插入的话，数据库就不用一遍又一遍地处理每一个文档的这种信息了。

批量插入用于应用程序中，比如一次插入数百个传感器采样点到分析集合。只有插入多个文档到一个集合的时候，这种方式才会有用，而不能用批量插入一次对多个集合执行操作。要是只是导入原始数据（例如，从数据 feed 或者 MySQL 中导入），可以使用命令行工具，如 `mongoimport`，而不是使用批量插入。另一方面，可以用它在存入 MongoDB 之前对数据做一些小的修整（转换日期成为日期类型，或添加自定义的 `"_id"`），所以批量插入对导入数据来说也是有用的。

当前版本的 MongoDB 消息长度最大是 16 MB，所以使用批量插入时还是有限制的。

3.1.2 插入：原理和作用

当执行插入的时候，使用的驱动程序会将数据转换成 BSON 的形式，然后将其送入数据库（关于 BSON，详见附录 C）。数据库解析 BSON，检验是否包含 `"_id"` 键并且文档不超过 4 MB，除此之外，不做别的数据验证，就只是简单地将文档原样存入数据库中。这会带来些或好或坏的影响，最明显的副作用就是允许插入无效的数据，而从好处看，它能让数据库更加安全，远离注入式攻击。

所有主流语言（也包括绝大部分非主流语言）的驱动会在传送数据之前进行一些数据的有效性检查（文档是否超长，是否包含非 UTF-8 字符，或者使用了未知类型）。要是对使用的驱动拿捏不准，可以在启动数据库服务器的时候使用 `--objcheck` 选项，这样服务器就会在插入之前先检查文档结构的有效性（当然这么做要牺牲些性能）。



大于 4 MB[1]（转换成 BSON）的文档是不能存入数据库的。这算是有点随意挑选的大小（可能以后会增加）。主要是避免不良的模式设计，保证稳定的性能。要查看 *doc* 文档转为 BSON 的大小（以字节为单位），在 shell 中运行 `Object.bsonsize(`*`doc`*`)` 即可。

4 MB 究竟是个多大空间呢，要知道整部《战争与和平》也才 3.14 MB。

MongoDB 在插入时并不执行代码，所以这块没有注入式攻击的可能。传统的注入式攻击对 MongoDB 来说是无效的，类似的注入式型攻击一般来说也是非常容易对抗的，何况插入对这种攻击天生免疫。

3.2 删除文档

现在数据库中有些数据，要删除它：

```
> db.users.remove()
```

译注1：MongoDB 1.8支持16 MB。

上述命令会删除 users 集合中所有的文档。但不会删除集合本身，原有的索引也会保留。

remove 函数可以接受一个查询文档作为可选参数。给定这个参数以后，只有符合条件的文档才被删除。例如，假设要删除 mailing.list 集合中所有 "opt-out" 为 true 的人：

```
> db.mailing.list.remove({"opt-out" : true})
```

删除数据是永久性的，不能撤销，也不能恢复。

删除速度

删除文档通常会很快，但是要清除整个集合，直接删除集合（然后重建索引）会更快。

例如，在 Python 中，使用如下方法插入一百万个虚拟元素：

```
for i in range(1000000):
    collection.insert({"foo": "bar", "baz": i, "z": 10 - i})
```

现在把刚插入的文档都删除，并记录花费的时间。首先来看一个简单的删除（remove）：

```
import time

from pymongo import Connection

db = Connection().foo
collection = db.bar

start = time.time()

collection.remove()
collection.find_one()

total = time.time() - start
print "%d seconds" % total
```

在 MacBook Air 笔记本电脑上，这段脚本输出“46.08 seconds”（46.08 秒）。

如果用 db.drop_collection("bar") 来代替 remove 和 find_one，只花了 .01 秒！速度提升相当明显，但也是有代价的：不能有任何限制条件。整个集合都被删除了，所有的索引也都不见了。

3.3 更新文档

文档存入数据库以后，就可以使用 update 方法来修改它。update 有两个参数，一个是查询文档，用来找出要更新的文档，另一个是修改器（modifier）文档，描述对找到的文档做哪些更改。

更新操作是原子的：若是两个更新同时发生，先到达服务器的先执行，接着执行另外一个。所以，互相有冲突的更新可以火速传递，并不会相互干扰：最后的更新会取得“胜利”。

3.3.1 文档替换

更新最简单的情形就是完全用一个新文档替代匹配的文档。这适用于模式结构发生了较大变化的时候。例如，要对下面的用户文档做一个比较大的调整：

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "name" : "joe",
    "friends" : 32,
    "enemies" : 2
}
```

想变成下面的样子：

```
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
    "username" : "joe",
    "relationships" :
        {
            "friends" : 32,
            "enemies" : 2
        }
}
```

可以用 update 来替换文档：

```
> var joe = db.users.findOne({"name" : "joe"});
> joe.relationships = {"friends" : joe.friends,"enemies" : joe.enemies};
{
    "friends" : 32,
    "enemies" : 2
}

> joe.username = joe.name;
"joe"
> delete joe.friends;
true
> delete joe.enemies;
true
> delete joe.name;
true
> db.users.update({"name" : "joe"}, joe);
```

现在，用 findOne 查看更新后的文档结构。

常见错误就是查询条件匹配了多个文档，然后更新的时候由于第二个参数的存在就产生重复的 "_id" 值。数据库会报错[1]，不做任何修改。

译注1：除了shell外、一般程序是不会报错的，除非用getLastError。

例如，有好几个文档都有相同的 "name"，但是我们没有意识到：

```
> db.people.find()
{"_id" : ObjectId("4b2b9f67a1f631733d917a7b"), "name" : "joe", "age" : 65},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7c"), "name" : "joe", "age" : 20},
{"_id" : ObjectId("4b2b9f67a1f631733d917a7d"), "name" : "joe", "age" : 49},
```

现在如果第二个 Joe 过生日，要增加 "age" 的值，可能会这么做：

```
> joe = db.people.findOne({"name" : "joe", "age" : 20});
{
    "_id" : ObjectId("4b2b9f67a1f631733d917a7c"),
    "name" : "joe",
    "age" : 20
}
> joe.age++;
> db.people.update({"name" : "joe"}, joe);
E11001 duplicate key on update
```

到底怎么了呢？当调用 update 时，数据库会查找一个与 {"name" : "joe"} 匹配的文档。找到的第一个就是那个 65 岁的 Joe。然后数据库试着用变量 joe 中的内容替换找到的文档，但是会发现集合里面已经有一个具有同样 "_id" 的文档。所以，更新就会失败，因为 "_id" 值必须唯一。为了避免这种情况，最好确保更新总是指定唯一文档，例如通过像 "_id" 这样的键来匹配。

3.3.2 使用修改器

通常文档只会有一部分要更新。利用原子的*更新修改器*，可以使得这种部分更新极为高效。更新修改器是种特殊的键，用来指定复杂的更新操作，比如调整、增加或者删除键，还可能是操作数组或者内嵌文档。

假设要在一个集合中放置网站的分析数据，每当有人访问页面的时候，就要增加计数器。可以使用更新修改器原子性地完成这个增加。每个 URL 及对应的访问次数都以如下的方式存储在文档中：

```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

每次有人访问页面，就通过 URL 找到该页面，并用 "$inc" 修改器增加 "pageviews" 的值。

```
> db.analytics.update({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
```

接着，执行一个 find 操作，会发现 "pageviews" 的值增加了 1。

```
> db.analytics.find()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```



Perl 和 PHP 程序员可能会想：要是用别的字母而不是 $，就更好了。在这两种语言里 $ 表示变量前缀，在双引号中的以 $ 开头的字符串都会被替换成变量的值。然而，MongoDB 一开始设计成 JavaScript 数据库，$ 在 JavaScript 中并没什么特殊含义，所以就这么用了。这的确是一个 MongoDB 的历史遗留问题。

Perl 和 PHP 程序员还是有些选择的。首先，可以转义 $："\$foo"。也可使用单引号 '$foo'，就不会有变量解释了。最后，这两种语言的驱动程序都可以不使用 $，而用自己的定义。在 Perl 中，设置 $MongoDB::BSON::char，在 PHP 中设置 php.ini 文件的 mongo.cmd_char，可以用 =、:、?，或者任何你觉得可以替代 $ 的字符都可以。比如，你选了～，就可以用～ inc 当做 \$inc，把～ gt 当做 \$gt。

尽量不要选择会出现在键名中的字符（_ 或者 x），也不要使用自身会转义的字符，这只会徒增烦恼（比如 \ 或者 Perl 中的 @）。

使用修改器时，"_id" 的值不能改变。（注意，整个文档替换时是可以改变 "_id" 的。）其他键值，包括其他唯一索引的键，都是可以更改的。

1. "$set"修改器入门

"$set" 用来指定一个键的值。如果这个键不存在，则创建它。这对更新模式或者增加用户定义键来说非常方便。例如，用户资料存储在下面这样的文档里：

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

非常简要的一段用户信息。要想添加喜欢的书籍进去，可以使用 "$set"：

```
> db.users.update({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "war and peace"}})
```

之后文档就有了“favorite book”键。

```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin",
    "favorite book" : "war and peace"
}
```

要是用户觉得喜欢的其实是另外一本书，`"$set"` 又能帮上忙了：

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" : "green eggs and ham"}})
```

用 `"$set"` 甚至可以修改键的数据类型。例如，如果用户又觉得喜欢的是一堆书，就可以将“favorite book”键的值变成一个数组：

```
> db.users.update({"name" : "joe"},
... {"$set" : {"favorite book" :
...     ["cat's cradle", "foundation trilogy", "ender's game"]}})
```

如果用户突然发现自己不爱读书，可以用 `"$unset"` 将键完全删除：

```
> db.users.update({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

现在这个文档就和例子开始的时候一样了。

也可以用 `"$set"` 修改内嵌文档：

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.update({"author.name" : "joe"}, {"$set" : {"author.
name" : "joe schmoe"}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe schmoe",
```

```
            "email" : "joe@example.com"
        }
    }
```

增加、修改或删除键的时候，应该使用 $ 修改器。要把 "foo" 的值设为 "bar"，常见的错误做法如下：

```
> db.coll.update(criteria, {"foo" : "bar"})
```

这会事与愿违。实际上这会将整个文档用 {"foo":"bar"} 替换掉。一定要使用以 $ 开头的修改器来修改键 / 值对。

2. 增加和减少

"$inc" 修改器用来增加已有键的值，或者在键不存在时创建一个键。对于分析数据、因果关系、投票或者其他有变化数值的地方，使用这个都会非常方便。

假如建立了一个游戏集合，将游戏和变化的分数都存储在里面。比如用户玩弹球游戏（pinball），可以插入一个包含游戏名和玩家的文档来标识不同的游戏：

```
> db.games.insert({"game" : "pinball", "user" : "joe"})
```

要是小球撞到了砖块，就会给玩家加分。分数可以随便给，这里就把玩家得分基数约定成 50 好了。使用 "$inc" 修改器给玩家加 50 分：

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 50}})
```

更新后，可以看到：

```
> db.games.findOne()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 50
}
```

分数键（score）原来并不存在，所以 "$inc" 创建了这个键，并把值设定成增加量：50。

如果小球落入加分区，要加 10 000 分。只要给 "$inc" 传递一个不同的值就好了：

```
> db.games.update({"game" : "pinball", "user" : "joe"},
... {"$inc" : {"score" : 10000}})
```

现在来看看结果：

```
> db.games.find()
{
     "_id" : ObjectId("4b2d75476cc613d5ee930164"),
     "game" : "pinball",
     "name" : "joe",
     "score" : 10050
}
```

"score" 键存在并有数字类型的值，所以服务器就把这个值加了 10 000。

"$inc" 与 "$set" 的用法类似，就是专门来增加（和减少）数字的。"$inc" 只能用于整数、长整数或双精度浮点数。要是用在其他类型的数据上就会导致操作失败。其中包括很多语言会自动转换成数字的类型，例如 null、布尔类型或数字构成的字符串。

```
> db.foo.insert({"count" : "1"})
> db.foo.update({}, {$inc : {count : 1}})
Cannot apply $inc modifier to non-number
```

另外，"$inc" 键的值必须为数字。不能使用字符串、数组或其他非数字的值。否则就会提示“Modifier "$inc" allowed for numbers only”（修改器 "$inc" 只允许使用数字）这样的错误。要修改其他类型，应该使用 "$set" 或者一会儿要讲到的数组修改器。

3. 数组修改器

有一类很好的修改器可用于操作数组。数组是常用且非常有用的数据结构：它们不仅是可通过索引进行引用的列表，而且还可以作为集合来用。

数组操作，顾名思义，只能用在值为数组的键上。例如不能对整数做 push，也不能对字符串做 pop。使用 "$set" 或 "$inc" 来修改标量值。

如果指定的键已经存在，"$push" 会向已有的数组末尾加入一个元素，要是没有就会创建一个新的数组。例如，假设要存储博客文章，要添加一个包含一个数组的 "comments"（评论）键。可以向还不存在的 "comments" 数组 push 一个评论，这个数组会被自动创建，并加入评论：

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
> db.blog.posts.update({"title" : "A blog post"}, {$push : {"comments"
:
... {"name" : "joe", "email" : "joe@example.com", "content" : "nice
     post."}}})
> db.blog.posts.findOne()
{
```

```
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

要是还想添加一条评论，可以接着使用 "$push"：

```
> db.blog.posts.update({"title" : "A blog post"},{$push : {"comments":
... {"name" : "bob", "email" : "bob@example.com", "content" : "good
      post."}}})
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

经常会有这种情况，如果一个值不在数组里面就把它加进去。可以在查询文档中用 "$ne" 来实现。例如，要是作者不在引文列表中就添加进去，可以这么做：

```
> db.papers.update({"authors cited" : {"$ne" : "Richie"}},
... {$push : {"authors cited" : "Richie"}})
```

也可以用 "$addToSet" 完成同样的事，要知道有些情况 "$ne" 根本行不通，有些时候更适合用 "$addToSet"。

例如，有一个表示用户的文档，已经有了电子邮件地址信息：

```
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
```

```
            "joe@example.com",
            "joe@gmail.com",
            "joe@yahoo.com"
        ]
    }
```

当添加新的地址时，用 "$addToSet" 可以避免重复：

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@gmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
    ]
}
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
... {"$addToSet" : {"emails" : "joe@hotmail.com"}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
    ]
}
```

将 "$addToSet" 和 "$each" 组合起来，可以添加多个不同的值，而用 "$ne" 和 "$push" 组合就不能实现。例如，想一次添加多个邮件地址，就可以使用这些修改器：

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")},
                  {"$addToSet" :
... {"emails" : {"$each" : ["joe@php.net", "joe@example.com", "joe@
                  python.org"]}}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
```

```
            "joe@python.org"
        ]
    }
```

有几个从数组中删除元素的方法。若是把数组看成队列或者栈，可以用 "$pop"，这个修改器可以从数组任何一端删除元素。{$pop : {*key* : 1}} 从数组末尾删除一个元素，{$pop : {*key* : -1}} 则从头部删除。

有时需要基于特定条件来删除元素，而不仅仅是依据位置，"$pull" 可以做到。例如，有个待完成事项列表，顺序有些问题：

```
> db.lists.insert({"todo" : ["dishes", "laundry", "dry cleaning"]})
```

要是想把洗衣服（laundry）放到第一位，可以从列表中先删掉：

```
> db.lists.update({}, {"$pull" : {"todo" : "laundry"}})
```

通过查找，会看到只有两个元素了：

```
> db.lists.find()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "todo" : [
        "dishes",
        "dry cleaning"
    ]
}
```

"$pull" 会将所有匹配的部分删掉。对数组 [1, 1, 2, 1] 执行 pull 1，得到结果就是只有一个元素的数组 [2]。

4. 数组的定位修改器

若是数组有多个值，而我们只想对其中的一部分进行操作，这就需要一些技巧。有两种方法操作数组中的值：通过位置或者定位操作符（"$"）。

数组都是以 0 开头的，可以将下标直接作为键来选择元素。例如，这里有个文档，其中包含由内嵌文档组成的数组，比如包含评论的博客文章。

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b329a216cc613d5ee930192"),
    "content" : "...",
    "comments" : [
        {
            "comment" : "good post",
            "author" : "John",
            "votes" : 0
```

```
        },
        {
            "comment" : "i thought it was too short",
            "author" : "Claire",
            "votes" : 3
        },
        {
            "comment" : "free watches",
            "author" : "Alice",
            "votes" : -1
        }
    ]
}
```

如果想增加第一个评论的投票数量，可以这么做：

```
> db.blog.update({"post" : post_id},
... {"$inc" : {"comments.0.votes" : 1}})
```

但是很多情况下，不预先查询文档就不能知道要修改数组的下标。为了克服这个困难，MongoDB 提供了定位操作符 `"$"`，用来定位查询文档已经匹配的元素，并进行更新。例如，要是用户 John 把名字改成 Jim，就可以用定位符替换评论中的名字：

```
db.blog.update({"comments.author" : "John"},
... {"$set" : {"comments.$.author" : "Jim"}})
```

定位符只更新第一个匹配的元素。所以，如果 John 有不止一个评论，那么只有他的第一条评论中的名字会被更改。

5. 修改器速度

有的修改器运行比较快。`$inc` 能就地修改，因为不需要改变文档的大小，只需要将键的值修改一下，所以非常快。而数组修改器可能更改了文档的大小，就会慢一些（`"$set"` 能在文档大小不发生变化时立即修改，否则性能也会有所下降）。

MongoDB 预留了些补白给文档，来适应大小变化（事实上，系统会根据文档通常的大小变化情况来相应地调整补白的大小），但是要是超出了原来的空间，最后还是要分配一块新的空间。空间分配除了会减慢速度，同时会随着数组变长，MongoDB 需要更长的时间来遍历整个数组，对每个数组的修改也会慢下来。

用个简单 Python 程序就能验证速度的差异。这个程序插入一个键，并增加其值 100 000 次。

```
from pymongo import Connection

import time

db = Connection().performance_test
db.drop_collection("updates")
```

```
collection = db.updates

collection.insert({"x": 1})

# make sure the insert is complete before we start timing
collection.find_one()

start = time.time()

for i in range(100000):
    collection.update({}, {"$inc" : {"x" : 1}})

# make sure the updates are complete before we stop timing
collection.find_one()

print time.time() - start
```

在一台 MacBook Air 上，一共花了 7.33 秒。每秒有 13 000 多次更新（对于一台虚弱的小机器来说已经相当不错了）。现在看看 push 100 000 次的话，会有什么效果：

```
for i in range(100000):
    collection.update({}, {'$push' : {'x' : 1}})
```

程序花了 67.58 秒，也就是说每秒更新不到 1500 次。

`"$push"` 或者其他数组修改器是推荐使用的，有些场合还是十分必要的，但是一定要留心这种更新的利弊。要是 `"$push"` 成为瓶颈，可以将内嵌数组独立出来，放到单独一个集合里面。

3.3.3 upsert

`upsert` 是一种特殊的更新。要是没有文档符合更新条件，就会以这个条件和更新文档为基础创建一个新的文档。如果找到了匹配的文档，则正常更新。`upsert` 非常方便，不必预置集合，同一套代码可以既创建又更新文档。

让我们回过头看看那个记录网站页面访问次数的例子。要是没有 `upsert`，就得试着查询 URL，没有找到就得新建一个文档，找到的话就增加访问次数。要是把这个写成 JavaScript 程序（而不是用 mongo *scriptname.js* 来运行的一系列 shell 命令），会是如下这样的：

```
// check if we have an entry for this page
blog = db.analytics.findOne({url : "/blog"})

// if we do, add one to the number of views and save
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
```

```
}
// otherwise, create a new document for this page
else {
  db.analytics.save({url : "/blog", pageviews : 1})
}
```

这就是说如果有人访问页面，我们得去数据库打个来回，然后选择更新或者插入。要是多个进程同时运行这段代码，还得考虑对于给定 URL 不能同时插入文档的限制。

要是使用 upsert，既可以避免竞态问题，又可以缩减代码量（update 的第 3 个参数表示这是个 upsert）：

```
db.analytics.update({"url" : "/blog"}, {"$inc" : {"visits" : 1}}, true)
```

这行代码和之前的代码作用完全一样，但它更高效，并且是原子性的！创建新文档会将条件文档作为基础，然后将修改器文档应用于其上。例如，要是执行一个匹配键并增加对应键值的 upsert 操作，会在匹配的基础上进行增加：

```
> db.math.remove()
> db.math.update({"count" : 25}, {"$inc" : {"count" : 3}}, true)
> db.math.findOne()
{
    "_id" : ObjectId("4b3295f26cc613d5ee93018f"),
    "count" : 28
}
```

先是 remove 清空了集合，里面就没有文档了。upsert 创建一个键 "count" 的值为 25 的文档，随后将这个值加 3，最后得到 "count" 为 28 的文档。要是不开启 upsert 选项，{"count" : 25} 不会匹配到任何文档，也就没有任何更改。

要是将这个 upsert（条件为 {count:25}）再次运行，还会创建一个新文档。这是因为没有文档满足匹配条件（唯一的文档的 "count" 的值是 28）。

save Shell 帮助程序

save 是一个 shell 函数，可以在文档不存在时插入，存在时更新。它只有一个参数：文档。要是这个文档含有 "_id" 键，save 会调用 upsert。否则，会调用插入。程序员可以非常方便地使用这个函数在 shell 中快速修改文档。

```
> var x = db.foo.findOne()
> x.num = 42
42
> db.foo.save(x)
```

要是不用 save，最后一行可以像下面这样写，但很啰嗦：

```
db.foo.update({"_id" : x._id}, x).
```

3.3.4 更新多个文档

默认情况下，更新只能对符合匹配条件的第一个文档执行操作。要是有多个文档符合条件，其余的文档就没有变化。要使所有匹配到的文档都得到更新，可以设置 update 的第 4 个参数为 true。



今后可能会更改 update 的行为（服务器可能默认会更新所有匹配的文档，只有第 4 个参数为 false 才会只更新一个），所以建议每次都显式表明要不要做多文档更新。

这样不但更明确地指定了 update 的行为，还可以在默认参数发生变化时从容应对。

多文档更新对模式迁移非常有用，还可以在对特定用户发布新功能的时候使用。例如，假设要给所有在特定日期过生日的用户一份礼物，就可以使用多文档更新，将 "gift" 增加到他们的账号。

```
> db.users.update({birthday : "10/13/1978"},
... {$set : {gift : "Happy Birthday!"}}, false, true)
```

这样就给生日为 1978 年 10 月 13 日的所有用户文档添加了 "gift" 键。

想要知道多文档更新到底更新了多少文档，可以运行 getLastError 命令（或许叫做 "getLastOpStatus" 更为合适）。键 "n" 的值就是要的数字。

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

这里 "n" 为 5，说明有 5 个文档被更新了。"updatedExisting" 为 true，说明是对已有的文档进行更新。关于数据库命令及其作用的更多细节，可以参考第 7 章。

3.3.5 返回已更新的文档

用 getLastError 仅能获得有限的信息，并不能返回已更新的文档。这个可以通过 findAndModify 命令来做到。

findAndModify 的调用方式和普通的更新略有不同，还有点慢，这是因为它要等待数据库的响应。这对于操作查询以及执行其他需要取值和赋值风格的原子性操作来说是十分方便的。

假设我们有一个集合，其中包含以一定顺序运行的进程。其中每个进程都被表示为具有如下形式的文档：

```
{
    "_id" : ObjectId(),
    "status" : state,
    "priority" : N
}
```

"status" 是一个字符串，可以是 "READY"、"RUNNING" 或 "DONE"。要找到状态为 "READY" 的具有最高优先级的任务，运行进程函数，然后更新其状态为 "DONE"。将已经就绪的进程按照优先级排序，然后将优先级最高的进程的状态更新为 "RUNNING"。完成了以后，就把状态改为 "DONE"。就像下面这样：

```
ps = db.processes.find({"status" : "READY").sort({"priority" : -1}).
                         limit(1).next()
db.processes.update({"_id" : ps._id},{"$set" : {"status" : "RUNNING"}})
do_something(ps);
db.processes.update({"_id" : ps._id},{"$set" : {"status" : "DONE"}})
```

这个算法有问题，可能会导致竞态条件。假设有两个线程正在运行。A 线程读取了文档，B 线程在 A 状态改为 "RUNNING" 之前也读取了同一个文档，这样两个线程会运行相同的处理过程。虽然可以通过检查状态的方式来避免这一问题，但是十分麻烦：

```
var cursor = db.processes.find({"status" : "READY"}).sort({"priority" :
                                 -1}).limit(1);
while ((ps = cursor.next()) != null) {
    ps.update({"_id" : ps._id, "status" : "READY"},
              {"$set" : {"status" : "RUNNING"}});

    var lastOp = db.runCommand({getlasterror : 1});
    if (lastOp.n == 1) {
        do_something(ps);
        db.processes.update({"_id" : ps._id},{"$set" : {"status" : "DONE"}})
        break;
    }
    cursor = db.processes.find({"status" : "READY"}).sort({"priority" :
                                 -1}).limit(1);
}
```

这样也有问题。因为有先有后，很可能一个线程处理了所有任务，而另外一个就傻傻地呆在那里。A 线程可能会一直占用着进程，B 线程试着抢占失败后，就让 A 线程自己处理所有任务了。这种情况绝对适合用 findAndModify。这样就可以在一

个操作中返回结果并更新。具体步骤如下：

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}})
{
    "ok" : 1,
    "value" : {
        "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
        "priority" : 1,
        "status" : "READY"
    }
}
```

注意，返回文档中的状态仍然为 "READY"。先返回结果然后更新。要是再看看集合，会发现 "status" 已经更新成了 "RUNNING"：

```
> db.processes.findOne({"_id" : ps.value._id})
{
    "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
    "priority" : 1,
    "status" : "RUNNING"
}
```

这样的话，程序就变成了下面这样：

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "update" : {"$set" : {"status" : "RUNNING"}}).value
> do_something(ps)
> db.process.update({"_id" : ps._id}, {"$set" : {"status" : "DONE"}})
```

findAndModify 既有 "update" 键也有 "remove" 键。"remove" 键表示将匹配到的文档从集合里面删除。例如，现在不要更新状态了，而是直接删掉，就可以像下面这样：

```
> ps = db.runCommand({"findAndModify" : "processes",
... "query" : {"status" : "READY"},
... "sort" : {"priority" : -1},
... "remove" : true).value
> do_something(ps)
```

findAndModify 命令中每个键对应的值如下所示。

- findAndModify

  字符串，集合名。

- query

  查询文档，用来检索文档的条件。

- `sort`
  排序结果的条件。

- `update`
  修改器文档，对所找到的文档执行的更新。

- `remove`
  布尔类型，表示是否删除文档。

- `new`
  布尔类型，表示返回的是更新前的文档还是更新后的文档。默认是更新前的文档。

`"update"` 和 `"remove"` 必须有一个，也只能有一个。要是匹配不到文档，这个命令会返回一个错误。

这个命令有些限制。它一次只能处理一个文档，也不能执行 `upsert` 操作，只能更新已有文档。

相比普通更新来说，`findAndModify` 速度要慢一些。话虽这么说，它也不会太慢，大概耗时相当于一次查找、一次更新和一次 `getLastError` 顺序执行所需的时间。

3.4 瞬间完成

本章所讨论的 3 个操作（插入、删除和更新）都是瞬间完成的，这是因为它们都不需要等待数据库响应。这并不是异步操作，可以把这个想象成发出后就不再操心的动作（后面简称其为“离弦之箭”），客户端将文档发送给服务器后就立刻干别的了。客户端永远不会收到“好的，知道了”或者“有问题，能重新传送一遍吗”这类响应。

这个特点的优点很明显，速度快，这些操作都会非常快地执行，它只会受客户端发送的速度和网络速度的制约。通常会工作得很好，但有时也会出岔子：服务器崩溃了，网线被老鼠咬断了，数据中心被洪水淹了。在没有服务器的情况下，客户端还是会发送写操作到服务器的，完全不理会到底有没有服务器。这对有些应用是可以接受的，由于硬件故障丢失几秒钟的日志记录、用户点击或者分析数据没什么大不了的。但是对于另一些应用（如付费系统），这样就不好玩了。

3.4.1 安全操作

假设要完成一个电子商务系统。如果某人订购了某物，应用程序应该花点时间确保订单顺利。这就是为什么要给这些操作弄个“安全”版本，执行时检查到了错误还可以重来。



MongoDB 的开发者选择了不安全的版本作为默认选择，这是由于他们与关系型数据库打交道的经验所导致的。很多构建在关系型数据库之上的应用程序都根本不关心返回的代码，也不会检查返回码，但又得苦苦等待这个返回码，这会造成性能的极大下降。MongoDB 让用户来选择。这样，像一些收集日志记录或者实时数据分析的程序，就不用等待它们根本不在乎的返回码了。

安全的版本在执行完了操作后立即运行 `getLastError` 命令，来检查是否执行成功（详见 7.1 节有关命令的更多信息）。驱动程序会等待数据库响应，然后适当地处理错误，一般会抛出一个可被捕获的异常。这样，开发者就能用自己的语言以比较自然的方式捕获并处理数据库错误了。要是操作成功，`getLastError` 会给出额外的信息作为响应（例如，对于更新或删除操作，会给出受到影响的文档数量）。



`getLastError` 在增强安全性方面还包括查看操作是否成功地被复制这一功能。关于这一功能的细节，请参见 9.4.4 节。

“安全”的代价就是性能。即便忽略客户端处理异常的开销（不同语言的开销不一样，但一般都不是轻量级的），等待数据库响应本身的时间比只发送消息的时间多一个数量级。所以，应用程序需要权衡数据的重要性（以及丢失后的后果）及速度需求。



拿不准时，就采用安全操作。要是速度不够快，就让一些不太重要的操作变成离弦之箭。

具体来说：

- 如果不考虑安全性的话，就只用离弦之箭的方式；
- 想要活得稍长一点，就把重要的用户数据（帐号、信用卡号、电子邮件）用安全的方式操作，其余的数据就采用离弦之箭的方式；
- 如果你很谨慎，只用安全操作好了。不过要是应用程序自动生成数以百计的零散信息（例如，页面、用户或广告的统计信息）需要保存，这些还是可以用离弦之箭的方式对待。

3.4.2 捕获“常规”错误

安全操作不仅能对付前面那种世界末日的场景，也是一种调试数据库“奇怪”行为的好方法。即便安全操作最后会在生产环境中移除，但是在开发过程中还是应该大量地使用。这样可以避免很多常见的数据库使用错误，最常见的就是键重复的错误。

键重复错误经常发生在试图插入一个其 `"_id"` 值已被占用的文档。MongoDB 中不

允许在一个集合里面有多个 "_id" 值一样的文档。如果做的是安全插入，发生了键重复错误，安全检查会发现这个服务器错误，并抛出异常。在不安全模式下，数据库没有响应，所以可能根本就不知道插入失败了。

例如，在 shell 中，可以看到插入 "_id" 值相同的两个文档是行不通的：

```
> db.foo.insert({"_id" : 123, "x" : 1})
> db.foo.insert({"_id" : 123, "x" : 2})
E11000 duplicate key error index: test.foo.$_id_ dup key: { : 123.0 }
```

这时查看集合，会发现只有第一个文档成功插入了。注意，这种错误也可由唯一索引导致，并不一定都由 "_id" 引发。shell 总会检查错误，而驱动程序中检查是可选项。

3.5 请求和连接

数据库会为每一个 MongoDB 数据库连接创建一个队列，存放这个连接的请求。当客户端发送一个请求，会被放到队列的末尾。只有队列中的请求都执行完毕，后续的请求才会执行。所以从单个连接就可以了解整个数据库，并且它总是能读到自己写的东西。

注意，每个连接都有独立的队列，要是打开两个 shell，就有两个数据库连接。在一个 shell 中执行插入，之后在另一个 shell 中进行查询不一定能得到插入的文档。然而，在同一个 shell 中，插入后再进行查询是一定能查到的。手动复现这个行为并不容易，但是在繁忙的服务器上，交错的插入 / 查找就显得稀松平常了。当开发者用一个线程插入数据，用另一个线程检查是否成功插入时，就会经常遇到这种问题。有那么一两秒钟时间，好像根本就没插入数据，但随后数据又突然冒出来。

使用 Ruby、Python 和 Java 驱动程序时要特别注意这种行为，因为这几个语言的驱动程序都使用了连接池。为了提高效率，这些驱动程序都和服务器建立了多个连接（一个连接池），并将请求分散到这些连接中去。好在它们都提供一些机制来确保一系列的请求都由一个连接来处理。MongoDB wiki（http://dochub.mongodb.org/drivers/connections）上有不同语言连接池的详细信息。

第 4 章
查询

本章将详细介绍查询。主要会涵盖以下几个方面。

- 使用 find 或者 findOne 函数和查询文档对数据库执行查询。
- 使用 $ 条件查询实现范围、集合包含、不等式和其他查询。
- 有些查询用查询文档，甚至 $ 条件语句都不能表达。对于这种复杂的查询，可以使用 $where 子句，用强大的 JavaScript 来表达。
- 查询将会返回一个数据库游标，游标只有在你需要的时候才会惰性地批量返回文档。
- 还有很多针对游标执行的元操作，包括忽略一定数量的结果，或者限定返回结果的数量，还有对结果排序。

4.1 find简介

MongoDB 中使用 find 来进行查询。查询就是返回一个集合中文档的子集，子集合的范围从 0 个文档到整个集合。find 的第一个参数决定了要返回哪些文档，其形式也是一个文档，说明要执行的查询细节。

空的查询文档 {} 会匹配集合的全部内容。要是不指定查询文档，默认就是 {}。例如：

```
> db.c.find()
```

将返回集合 c 中的所有内容。

当我们开始向查询文档中添加键 / 值对时，就意味着限定了查找的条件。对于绝大多数类型来说，这种方式很简单明了。整数匹配整数，布尔类型匹配布尔类型，字符串匹配字符串。查询简单的类型，只要指定想要查找的值就好了，十分简单。例如，想

要查找所有 "age" 的值为 27 的文档，直接将这样的键 / 值对写进查询文档就好了：

```
> db.users.find({"age" : 27})
```

要是想匹配一个字符串，比如值为 "joe" 的 "username" 键，那么直接写就好了：

```
> db.users.find({"username" : "joe"})
```

可以通过向查询文档加入多个键 / 值对的方式来将多个查询条件组合在一起，会解释成“条件 1 AND 条件 2 AND…AND 条件 *N*”。例如，要想查询所有用户名为“joe”且年龄为 27 岁的用户，可以像下面这样：

```
> db.users.find({"username" : "joe", "age" : 27})
```

4.1.1 指定返回的键

有时并不需要将文档中所有键 / 值对都返回。遇到这种情况，可以通过 find（或者 findOne）的第二个参数来指定想要的键。这样做既会节省传输的数据量，又能节省客户端解码文档的时间和内存消耗。

例如，如果只对用户集合的 "username" 和 "email" 键感兴趣，可以使用如下查询返回这些键：

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

可以看到，"_id" 这个键总是被返回，即便是没有指定也一样。

也可以用第二个参数来剔除查询结果中的某个键 / 值对。例如，文档中有很多键，但是不希望结果中含有 "fatal_weakness" 键：

```
> db.users.find({}, {"fatal_weakness" : 0})
```

也可以用来防止返回 "_id"：

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

4.1.2 限制

查询使用上还是有些限制的。数据库所关心的查询文档的值必须是常量。（在你自己的

代码里可以是正常的变量）。也就是不能引用文档中其他键的值。例如，要想保持库存，有原库存 "in_stock" 和已出售 "num_sold" 两个键，想通过比较两者来查询：

```
> db.stock.find({"in_stock" : "this.num_sold"}) // 不可行
```

的确有办法实现类似的操作（详见 4.4 节），但通常可以略微修改一下文档结构，能通过普通查询来完成操作，这样性能也会有所改进。在这个例子中，可以用初始存货 "initial_stock" 和存货 "in_stock" 两个键来改写文档。这样，每当有人购买物品，就将 "in_stock" 减去 1。最后用一个简单的查询查看哪种商品已脱销：

```
> db.stock.find({"in_stock" : 0})
```

4.2 查询条件

查询不仅能像前面说的那样精确匹配，还能匹配更加复杂的条件，比如范围、OR 子句和取反。

4.2.1 查询条件

"$lt"、"$lte"、"$gt" 和 "$gte" 就是全部的比较操作符，分别对应 <、<=、> 和 >=。可以将其组合起来以便查找一个范围的值。例如，查询在 18~30 岁（含）的用户，就可以像下面这样：

```
> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
```

这样的范围查询对日期尤为有用。例如，要查找在 2007 年 1 月 1 日前注册的人，可以像下面这样：

```
> start = new Date("01/01/2007")
> db.users.find({"registered" : {"$lt" : start}})
```

精确匹配日期是徒劳的，因为日期只精确到毫秒。通常只是想得到关于一天、一周或者是一个月的数据，这样范围查询就很有必要了。

对于文档的键值不等于某个特定值的情况，就要使用另外一种条件操作符 "$ne" 了，它表示“不相等”。若是想要查询所有名字不为“joe”的用户，可以像下面这样查询：

```
> db.users.find({"username" : {"$ne" : "joe"}})
```

"$ne" 能用于所有类型的数据。

4.2.2 OR查询

MongoDB 中有两种方式进行 OR 查询。"$in" 可以用来查询一个键的多个值。

"$or" 更通用一些，用来完成多个键值的任意给定值。

对于单一键要是有多个值与其匹配的话，就要用 "$in" 加一个条件数组。例如，抽奖活动的中奖号码是 725、542 和 390。要找出全部这些中奖数据，可以构建如下查询：

```
> db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
```

"$in" 非常灵活，可以指定不同的类型的条件和值。例如，在逐步将用户的 ID 号迁移成用户名的过程中，要做兼顾二者的查询：

```
> db.users.find({"user_id" : {"$in" : [12345, "joe"]}})
```

这会匹配 "user_id" 等于 12345 的文档，也会匹配 "user_id" 等于 "joe" 的文档。

要是 "$in" 对应的数组只有一个值，那么和直接匹配这个值效果是一样的。例如，{ticket_no : {$in : [725]}} 和 {ticket_no : 725} 的效果一样。

与 "$in" 相对的是 "$nin"，将返回与数组中所有条件都不匹配的的文档。要是想返回所有没有中奖的人，就可以用如下方法进行查询：

```
> db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})
```

查询将会返回没有那些号码的人。

"$in" 能对单个键做 OR 查询，但要是想找到 "ticket_no" 为 725 或者 "winner" 为 true 的文档该怎么办呢？对于这种情况，应该使用 "$or"。"$or" 接受一个包含所有可能条件的数组作为参数。上面中奖的例子如果用 "$or" 改写会是下面这个样子的：

```
> db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : true}]})
```

"$or" 可以含有其他条件句。例如，如果想要将 "ticket_no" 与那 3 个值匹配上，外加 "winner" 键，就可以这么做：

```
> db.raffle.find({"$or" : [{"ticket_no" : {"$in" : [725, 542, 390]}},
                           {"winner" : true}]})
```

使用普通的 AND 型的查询时，总是想尽可能地用最少的条件来限定结果的范围 [1]。OR 型的查询正相反：第一个条件尽可能地匹配更多的文档，这样才是最为有效的。

4.2.3 $not

"$not" 是元条件句，即可以用在任何其他条件之上。例如，就拿取模运算符 "$mod" 来

译注1：也就是说将最严苛的条件放置在最前面。

说。"$mod" 会将查询的值除以第一个给定值，若余数等于第二个给定值则返回该结果：

```
> db.users.find({"id_num" : {"$mod" : [5, 1]}})
```

上面的查询会返回 "id_num" 值为 1、6、11、16 等的用户。但要是想返回 "id_num" 为 2、3、4、5、7、8、9、10、12 等的用户，就要用 "$not" 了：

```
> db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})
```

"$not" 与正则表达式联合使用的时候极为有用，用来查找那些与特定模式不符的文档（4.3.2 节会详细介绍其用法）。

4.2.4　条件句的规则

如果比较一下上一章的更新修改器和前面的查询文档，会发现以 $ 开头的键处在不同的位置。在查询中，"$lt" 在内层文档，而更新中 "$inc" 则是外层文档的键。基本可以肯定：条件句是内层文档的键，而修改器则是外层文档的键。

可对一个键应用多个条件。例如，要查找年龄为 20~30 的所有用户，可以在 "age" 键上使用 "$gt" 和 "$lt"：

```
> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})
```

一个键可以有多个条件，但是一个键不能对应多个更新修改器。例如，修改器文档不能同时含有 {"$inc" : {"age" : 1}, "$set" : {age : 40}}，因为修改了 "age" 两次。但是对于查询条件句就没有这种限定。

4.3　特定于类型的查询

如第 2 章所述，MongoDB 的文档可以使用多种类型的数据。其中有一些在查询的时候会有特别的表现。

4.3.1　null

null 就有点奇怪。它确实能匹配自身，所以要是有一个包含如下文档的集合：

```
> db.c.find()
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

就可以按照预期的方式查询 "y" 键为 null 的文档：

```
> db.c.find({"y" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
```

但是，null 不仅仅匹配自身，而且匹配“不存在的”。所以，这种匹配还会返回缺少这个键的所有文档：

```
> db.c.find({"z" : null})
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{ "_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{ "_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
```

如果仅仅想要匹配键值为 null 的文档，既要检查该键的值是否为 null，还要通过 "$exists" 条件判定键值已经已存在：

```
> db.c.find({"z" : {"$in" : [null], "$exists" : true}})
```

不幸的是，没有 "$eq" 操作符，所以看上去有些费解，但是只有一个元素的 "$in" 操作符效果是一样的。

4.3.2 正则表达式

正则表达式能够灵活有效地匹配字符串。例如，想要查找所有名为 Joe 或者 joe 的用户，就可以使用正则表达式执行忽略大小写的匹配：

```
> db.users.find({"name" : /joe/i})
```

系统可以接受正则表达式标识（i），但不是一定要有。现在匹配了各种大小写组合形式的 joe，要是还要匹配各种大小写组合形式的 joey，就可以略微修改一下正则表达式：

```
> db.users.find({"name" : /joey?/i})
```

MongoDB 使用 Perl 兼容的正则表达式（PCRE）库来匹配正则表达式，PCRE 支持的正则表达式语法都能被 MongoDB 所接受。建议在查询中使用正则表达式前，先在 JavaScript shell 中检查一下语法，确保匹配与设想的一致。



MongoDB 可以为前缀型正则表达式（比如 /^joey/）查询创建索引，所以这种类型的查询会非常高效。

正则表达式也可以匹配自身。虽然几乎没有人直接将正则表达式插入到数据库中，但要是万一这么做了，也是可以用自身匹配的：

```
> db.foo.insert({"bar" : /baz/})
> db.foo.find({"bar" : /baz/})
```

```
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "bar" : /baz/
}
```

4.3.3 查询数组

查询数组中的元素也是非常容易的。数组绝大多数情况下可以这样理解：每一个元素都是整个键的值。例如，如果数组是一个水果清单，比如下面这样：

```
> db.food.insert({"fruit" : ["apple", "banana", "peach"]})
```

下面的查询：

```
> db.food.find({"fruit" : "banana"})
```

会成功匹配该文档。这个查询好比我们用了一个如下的文档（不合法的）进行的查询：`{"fruit" : "apple","fruit" : "banana", "fruit" : "peach"}`。

1. $all

如果需要通过多个元素来匹配数组，就要用 `"$all"` 了。这样就会匹配一组元素。例如，假设创建包含 3 个元素的如下集合：

```
> db.food.insert({"_id" : 1, "fruit" : ["apple", "banana", "peach"]})
> db.food.insert({"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]})
> db.food.insert({"_id" : 3, "fruit" : ["cherry", "banana", "apple"]})
```

要找到既有 `"apple"` 又有 `"banana"` 的文档，就得用 `"$all"` 来查询：

```
> db.food.find({fruit : {$all : ["apple", "banana"]}})
    {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
    {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}
```

顺序无关紧要。注意，第二个结果中 `"banana"` 在 `"apple"` 之前。要是对只有一个元素的数组使用 `"$all"`，就和不用 `"$all"` 一样了。例如，`{fruit:{$all:['apple']}` 和 `{fruit:'apple'}` 的查询效果是等价的。

也可以使用完整的数组精确匹配。但是，精确匹配对于有缺少或者冗余元素的情况就不大灵了。例如，下面的方法会匹配之前的第一个文档：

```
> db.food.find({"fruit" : ["apple", "banana", "peach"]})
```

但是下面这个就不会匹配：

```
> db.food.find({"fruit" : ["apple", "banana"]})
```

这个亦不会匹配：

```
> db.food.find({"fruit" : ["banana", "apple", "peach"]})
```

要是想查询数组指定位置的元素，则需使用 *key.index* 语法指定下标，例如：

```
> db.food.find({"fruit.2" : "peach"})
```

数组下标都是从 0 开始的，所以上面的表达式会用数组的第 3 个元素和 `"peach"` 匹配。

2. $size

`"$size"` 对于查询数组来说也是意义非凡，顾名思义，可以用其查询指定长度的数组。见下面的例子：

```
> db.food.find({"fruit" : {"$size" : 3}})
```

一种常见的查询需求就是需要一个长度范围。`"$size"` 并不能与其他查询子句组合（比如 `"$gt"`），但是这种查询可以通过在文档中添加一个 `"size"` 键的方式来实现。这样每一次向指定数组添加元素的时候，同时增加 `"size"` 的值。原来这样的更新：

```
> db.food.update({"$push" : {"fruit" : "strawberry"}})
```

就会变成下面这样：

```
> db.food.update({"$push" : {"fruit" : "strawberry"},"$inc" : {"size" : 1}})
```

增加的操作非常快，所以对性能的影响微乎其微。这样存储文档后，就可以像下面这样查询了：

```
> db.food.find({"size" : {"$gt" : 3}})
```

不幸的是，这种技巧并不能与 `"$addToSet"` 操作符同时使用。

3. $slice操作符

本章前面已经提及，`find` 的第二个参数是可选的，可以指定返回哪些键。`"$slice"` 返回数组的一个子集合。

例如，假设现在有一个博客文章的文档，要想返回前 10 条评论，可以：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
```

也可以返回后 10 条评论，只要用 -10 就可以了：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
```

"$slice" 也可以接受偏移值和要返回的元素数量，来返回中间的结果：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})
```

这个操作会跳过前 23 个元素，返回第 24 个 ~ 第 33 个元素。如果数组不够 33 个元素，则返回第 23 个元素后面的所有元素。

除非特别声明，否则使用 "$slice" 时将返回文档中的所有键。别的键说明符都是默认不返回未提及的键，这点与 "$slice" 不太一样。例如，有如下的博客文章文档：

```
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        },
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        },
    ]
}
```

并且我们用 "$slice" 来获取最后一条评论，可以这样：

```
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -1}})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "bob",
            "email" : "bob@example.com",
            "content" : "good post."
        }
    ]
}
```

"title" 和 "content" 都被返回了，即便是并没有显式地出现在键说明符中。

4.3.4 查询内嵌文档

有两种方法查询内嵌文档：查询整个文档，或者只针对其键 / 值对进行查询。

查询整个内嵌文档与普通查询完全相同。例如，有如下文档：

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },
    "age" : 45
}
```

要查寻姓名为Joe Schmoe的人可以这样：

```
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
```

然而，如果Joe决定添加一个代表中间名的键，这个查询就不好用了，因为那样就不匹配整个内嵌文档了。这种查询还是与顺序相关的，`{"last" : "Schmoe", "first" : "Joe"}`就什么都匹配不到。

如果允许的话，通常只针对内嵌文档的特定键值进行查询才是比较好的做法。这样，即便数据模式改变，也不会导致所有查询因为要精确匹配而一下子都挂掉。我们可以使用点表示法查询内嵌的键：

```
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})
```

现在，如果Joe增加了更多的键，这个查询依然会匹配他的姓和名。

这种点表示法是查询文档区别于其他文档的主要特点。查询文档可以包含点，来表达“深入内嵌文档内部”的意思。点表示法也是待插入的文档不能包含“.”的原因。将键作为URL保存的时候经常会遇到此类问题。一种解决方法就是在插入前或者提取后执行一个全局替换，将“.”替换成一个URL中的非法字符。

当文档结构变得更加复杂以后，内嵌文档的匹配需要些许技巧。例如，假设有博客文章若干，要找到由Joe发表的5分以上的评论。博客文章的结构如下例所示：

```
> db.blog.find()
{
    "content" : "...",
    "comments" : [
        {
            "author" : "joe",
            "score" : 3,
            "comment" : "nice post"
        },
        {
            "author" : "mary",
            "score" : 6,
            "comment" : "terrible post"
        }
    ]
}
```

不能直接用`db.blog.find({"comments":{"author":"joe","score":{"$gte":5}}})`

来查寻。内嵌文档匹配要求整个文档完全匹配，而这不会匹配 `"comment"` 键。使用 `db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}})` 同样也不会达到目的。因为符合 author 条件的评论和符合 score 条件的评论可能不是同一条评论。也就是说，会返回刚才显示的那个文档。因为 `"author" : "joe"` 在第一条评论中匹配了，`"score" : 6` 在第二条评论中匹配了。

要正确地指定一组条件，而不用指定每个键，要使用 `"$elemMatch"`。这种模糊的命名条件句能用来部分指定匹配数组中的单个内嵌文档的限定条件。所以正确的写法应该是这样的：

```
> db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe",
                                               "score" : {"$gte" : 5}}}})
```

`"$elemMatch"` 将限定条件进行分组，仅当需要对一个内嵌文档的多个键操作时才会用到。

4.4 $where查询

键 / 值对是很有表现力的查询方式，但是依然有些需求它无法表达。当其他方法都败下阵的时候，就轮到 `"$where"` 子句了，用它可以执行任意 JavaScript 作为查询的一部分。这就使得查询能做（几乎）任何事情。

最典型的应用就是比较文档中的两个键的值是否相等。例如，有个条目列表，如果其中的两个值相等则返回文档。请看如下示例：

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

第二个文档中，`"spinach"` 和 `"watermelon"` 的值相同，所以需要返回该文档。MongoDB 似乎永远不会提供一个 $ 条件符来做这个，所以只能用 `"$where"` 子句借助 JavaScript 来完成了：

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...            return true;
...         }
...     }
... }
... return false;
... }});
```

如果函数返回 `true`，文档就做为结果的一部分被返回；如果为 `false`，则不然。

前面用的是一个函数，也可以用一个字符串来指定 "$where" 查询。下面两种表达是完全等价的：

```
> db.foo.find({"$where" : "this.x + this.y == 10"})
> db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})
```

不是非常必要时，一定要避免使用 "$where" 查询，因为它们在速度上要比常规查询慢很多。每个文档都要从 BSON 转换成 JavaScript 对象，然后通过 "$where" 的表达式来运行。同样还不能利用索引。所以，只在走投无路时才考虑 "$where" 这种用法。将常规查询作为前置过滤，与 "$where" 组合使用可以不牺牲性能。如果可能的话，用索引根据非 "$where" 子句进行过滤，"$where" 只用于对结果进行调优。

另一种复杂查询的方式是利用 MapReduce，下一章会进行介绍。

4.5 游标

数据库使用*游标*来返回 find 的执行结果。客户端对游标的实现通常能够对最终结果进行有效的控制。可以限制结果的数量，略过部分结果，根据任意方向任意键的组合对结果进行各种排序，或者是执行其他一些功能强大的操作。

要想从 shell 中创建一个游标，首先要对集合填充一些文档，然后对其执行查询，并将结果分配给一个局部变量（用 var 声明的变量就是局部变量）。这里，先创建一个简单的集合，而后做个查询，并用 cursor 变量保存结果：

```
> for(i=0; i<100; i++) {
... db.c.insert({x : i});
... }
> var cursor = db.collection.find();
```

这么做的好处是一次可以查看一条结果。如果将结果放在全局变量或者就没有放在变量中，MongoDB shell 会自动迭代，自动显示最开始的若干文档。也就是在这之前我们看到的种种例子，一般大家只想通过 shell 看看集合里面有什么，而不是想在其中实际运行程序，这样设计也就很合适。

要迭代结果，可以使用游标的 next 方法。也可以使用 hasNext 来查看有没有其他结果。典型的结果遍历如下：

```
> while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
```

cursor.hasNext() 检查是否有后续结果存在，然后用 cursor.next() 将其获得。

游标类还实现了迭代器接口，所以可以在 foreach 循环中使用。

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
... print(x.name);
... });
adam
matt
zak
```

当调用 find 的时候，shell 并不立即查询数据库，而是等待真正开始要求获得结果的时候才发送查询，这样在执行之前可以给查询附加额外的选项。几乎所有游标对象的方法都返回游标本身，这样就可以按任意顺序组成方法链。例如，下面几种表达是等价的：

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

此时，查询还没有执行，所有这些函数都只是构造查询。现在，假设我们执行如下操作：

```
> cursor.hasNext()
```

这时，查询被发往服务器。shell 立刻获取前 100 个结果或者前 4 MB 数据（两者之中较小者），这样下次调用 next 或者 hasNext 时就不必兴师动众跑到服务器上去了。客户端用光了第一组结果，shell 会再一次联系数据库，并要求更多的结果。这个过程一直会持续到游标耗尽或者结果全部返回。

4.5.1 limit、skip和sort

最常用的查询选项就是限制返回结果的数量，忽略一定数量的结果并排序。所有这些选项一定要在查询被派发到服务器之前添加。

要限制结果数量，可在 find 后使用 limit 函数。例如，只返回 3 个结果，可以这样：

```
> db.c.find().limit(3)
```

要是匹配的结果不到 3 个，则返回匹配数量的结果。limit 指定的是上限，而非下限。

skip 与 limit 类似：

```
> db.c.find().skip(3)
```

上面的操作会略过前三个匹配的文档，然后返回余下的文档。如果集合里面能匹配的文档少于 3 个，则不会返回任何文档。

sort 用一个对象作为参数：一组键 / 值对，键对应文档的键名，值代表排序的方向。排序方向可以是 1（升序）或者 -1（降序）。如果指定了多个键，则按照多个键的顺

序逐个排序。例如，要按照 "username" 升序及 "age" 降序排序，可以这样写：

```
> db.c.find().sort({username : 1, age : -1})
```

这 3 个方法可以组合使用。这对于分页非常有用。例如，你有个在线商店，有人想搜索 mp3。若是想每页返回 50 个结果，而且按照价格从高到低排序，可以这样写：

```
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1})
```

点击“下一页”可以看到更多的结果，通过 skip 也可以非常简单地实现，只需要略过前 50 个结果就好了（已经在第一页显示了）：

```
> db.stock.find({"desc" : "mp3"}).limit(50).skip(50).sort({"price" : -1})
```

然而，略过过多的结果会导致性能问题，所以建议尽量避免。

比较顺序

MongoDB 处理不同类型的数据是有一个顺序的。有时候一个键的值可能是多种类型的，例如，整数和布尔类型，或者字符串和 null。如果对这种混合类型的键排序，其排序顺序是预先定义好的。从小到大，其顺序如下：

(1) 最小值
(2) null
(3) 数字（整型、长整型、双精度）
(4) 字符串
(5) 对象 / 文档
(6) 数组
(7) 二进制数据
(8) 对象 ID
(9) 布尔型
(10) 日期型
(11) 时间戳
(12) 正则表达式
(13) 最大值

4.5.2 避免使用skip略过大量结果

用 skip 略过少量的文档还是不错的。但是要是数量非常多的话，skip 就会变得很慢（几乎每个数据库都有这个问题，不仅仅是 MongoDB），所以要尽量避免。通常可以向文档本身内置查询条件，来避免过大的 skip，或者利用上次的结果来计算下一次查询。

1. 不用skip对结果分页

最简单的分页方法就是用 limit 返回结果的第一页，然后将每个后续页面作为相对于开始的偏移量返回。

```
> // do not use: slow for large skips
> var page1 = db.foo.find(criteria).limit(100)
> var page2 = db.foo.find(criteria).skip(100).limit(100)
> var page3 = db.foo.find(criteria).skip(200).limit(100)
...
```

然而，一般来讲可以找到一种方法实现不用 skip 的分页，这取决于查询本身。例如，要按照 "date" 降序显示文档。可以用如下方式获取结果的第一页：

```
> var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

然后，可以利用最后一个文档中 "date" 的值作为查询条件，来获取下一页：

```
var latest = null;

// display first page
while (page1.hasNext()) {
    latest = page1.next();
    display(latest);
}

// get next page
var page2 = db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

这样查询中就没有 skip 了。

2. 随机选取文档

从集合里面随机挑选一个文档算是个常见问题。最笨的（也是很慢的）做法就是先计算文档总数，然后选择一个从 0 到文档数量之间的随机数，利用 find 做一次查询，略过这个随机数那么多的文档，这个随机数的取值范围为 0 到集合中文档的总数。

```
> // do not use
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

这种选取随机文档的做法实在是低效：首先得计算总数（要是有查询条件就会很费时），然后大量的 skip 也会非常耗时。

略微动动脑筋，从集合里面查找一个随机元素还是有好得多的办法的。秘诀就是在插入文档时给每个文档都添加一个额外的随机键。例如在 shell 中，可以用 Math.random()（产生一个 0~1 的随机数）：

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
```

这样，想要从集合中查找一个随机文档，只要计算个随机数并将其作为查询条件就好了，完全不用 skip：

```
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
```

也有偶尔遇到随机数比所有集合里面存着的随机值大的情况[1]，这时就没有结果返回了。那就换个方向，这样就万事大吉了：

```
> if (result == null) {
... result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

要是集合里面本就没有文档，则会返回 null，这说得通。

这种技巧还可以和其他各种复杂的查询一同使用，仅需要确保有包含随机键的索引即可。例如，想随机找一个加州的水暖工，可以对 "profession"、"state" 和 "random" 建立索引：

```
> db.people.ensureIndex({"profession" : 1, "state" : 1, "random" : 1})
```

这样就能很快得出一个随机结果了（关于索引，详见第 5 章）。

4.5.3 高级查询选项

查询分为包装的和普通的两类。普通的查询就像下面这个：

```
> var cursor = db.foo.find({"foo" : "bar"})
```

有几个选项用于包装查询。例如，假设我们执行一个排序：

```
> var cursor = db.foo.find({"foo" : "bar"}).sort({"x" : 1})
```

实际情况不是将 {"foo" : "bar"} 作为查询直接发送给数据库，而是将查询包装在一个更大的文档中。shell 会把查询从 {"foo" : "bar"} 转换成 {"$query" :{"foo" : "bar"}, "$orderby" : {"x" : 1}}。

绝大多数驱动程序有些辅助措施向查询添加各种选项。下面列举了其他一些有用的选项。

- `$maxscan : `*integer*

译注1：还会有极少数相等的时候，$lte可能更加严谨。

指定查询最多扫描的文档数量。

- `$min` : *document*

查询的开始条件。

- `$max` : *document*

查询的结束条件。

- `$hint` : *document*

指定服务器使用哪个索引进行查询。

- `$explain` : *boolean*

获取查询执行的细节（用到的索引、结果数量、耗时等），而并非真正执行查询。

- `$snapshot` : *boolean*

确保查询的结果是在查询执行那一刻的一致快照。详见下一节。

4.5.4 获取一致结果

数据处理通常的一种做法就是先把数据从 MongoDB 中取出来，然后经过某种变换，最后再存回去：

```
cursor = db.foo.find();

while (cursor.hasNext()) {
    var doc = cursor.next();
    doc = process(doc);
    db.foo.save(doc);
}
```

结果比较少时还好，文档较多时就玩不转了。为什么呢？想象一下文档究竟是如何存储的吧。可以将集合看做一大堆文档，看上去就像图 4-1。雪花代表文档，因为每一个文档都是美丽且唯一的。

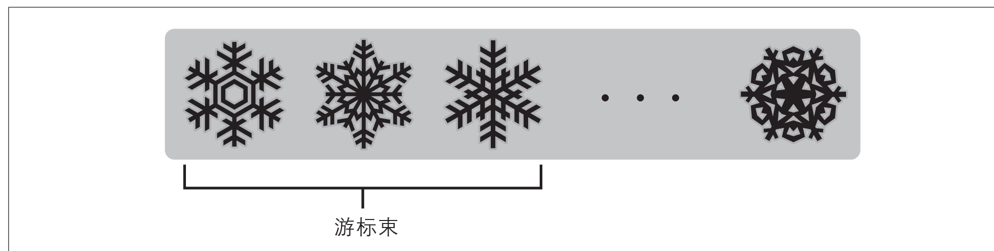


图 4-1：待查询的集合

这样，做查找的时候，从集合的开头返回结果，并向右移动。程序获取前 100 个文档并处理。当要将其保存回数据库时，如果文档体积增加而预留空间不足，就像图 4-2，则需要将其移动。通常会将其挪至集合的末尾处（如图 4-3 所示）。

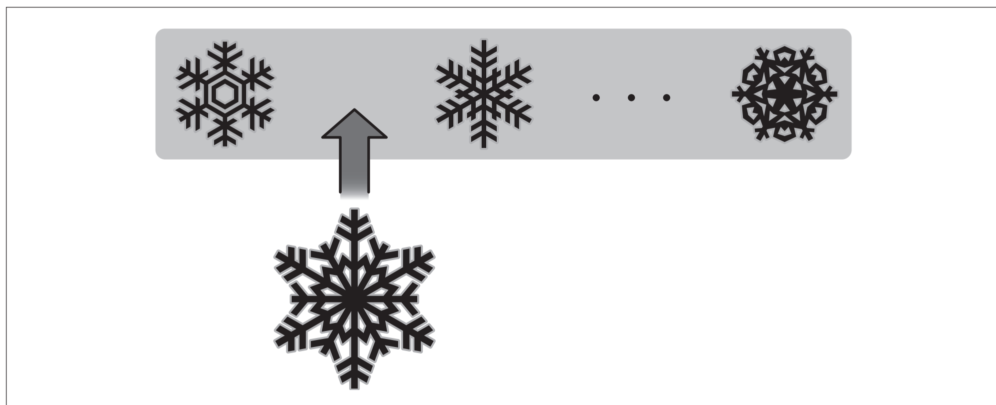


图 4-2：变大的文档已经超过了原来分配的空间

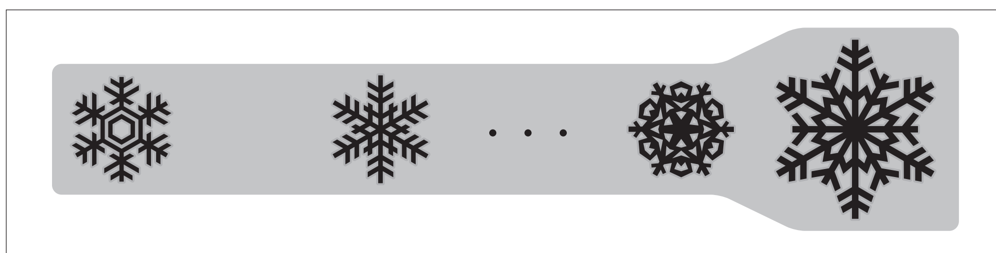


图 4-3：MongoDB 会移动不能放在原处的新文档

接着，程序继续获取大量的文档。如此往复，结果返回了已经被挪动的文档（如图 4-4 所示）。

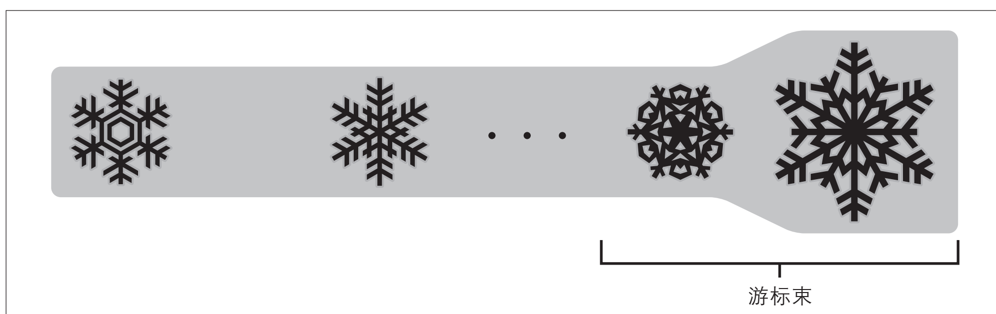


图 4-4：游标可能会返回那些已经被挪动的文档

应对这个问题的方法就是对查询进行快照。如果使用了 "$snapshot" 选项，查询就是针对不变的集合视图运行的。所有返回一组结果的查询实际上都进行了快照。不一致只在游标等待结果时集合内容被改变的情况下发生。

4.6 游标内幕

看待游标有两种角度：客户端的游标以及客户端游标表示的数据库游标。前面讨论的都是客户端的游标，接下来简要看看服务器端发生了什么。

在服务器端，游标消耗内存和其他资源。游标遍历尽了结果以后，或者客户端发来消息要求终止，数据库将会释放这些资源。释放的资源可以被数据库换作他用，这是非常有益的，所以要尽量保证尽快释放游标（在合理的前提下）。

还有一些情况导致游标终止（随后被清理）。首先，当游标完成匹配结果的迭代时，它会清除自身。另外，当游标在客户端已经不在作用域内了，驱动会向服务器发送专门的消息，让其销毁游标。最后，即便用户也没有迭代完所有结果，并且游标也还在作用域中，10 分钟不使用，数据库游标也会自动销毁。

这种“超时销毁”的行为是我们希望的：极少有应用程序希望用户花费数分钟坐在那里等待结果。然而，的确有些时候希望游标持续的时间长一些。若是如此的话，多数驱动程序都实现了一个叫 immortal 的函数，或者类似的机制，来告知数据库不要让游标超时。如果关闭了游标的超时时间，则一定要在迭代完结果后将其关闭，否则它会一直在数据库中消耗服务器资源。

第 5 章
索引

索引就是用来加速查询的。数据库索引与书籍的索引类似：有了索引就不需要翻遍整本书，数据库则可以直接在索引中查找，使得查找速度能提高几个数量级。在索引中找到条目以后，就可以直接跳转到目标文档的位置。

让这个比喻走个极端，可以说创建数据库索引就像确定如何组织书的索引一样。但你的优势是知道今后会做何种查询，以及哪些内容需要快速查找。比如，所有的查询都包括 "date" 键，那么很可能（至少）需要建立一个关于 "date" 的索引。如果要查询用户名，则不必索引 "user_num" 键，因为根本不会对其进行查询。

5.1 索引简介

要掌握如何为查询配置最佳索引会有些难度，但却是非常值得努力去做。有时候花费数分钟的查询，如果配合适当的索引可能会即刻完成。



MongoDB 的索引几乎与传统的关系型数据库索引一模一样，所以如果已经掌握了那些技巧，则可以跳过本节的语法说明。后面会有些索引的基础知识，但一定要记住这里涉及的只是冰山一角，绝大多数优化 MySQL/Oracle/SQLite 索引的技巧也同样适用于 MongoDB。

现在要依照某个键进行查找：

```
> db.people.find({"username" : "mark"})
```

当查询中仅使用一个键时，可以对该键建立索引，以提高查询速度。本例中，对 "username" 建立索引。创建索引要使用 ensureIndex 方法：

```
> db.people.ensureIndex({"username" : 1})
```

对于同一个集合，同样的索引只需要创建一次。反复创建是徒劳的。

对某个键创建的索引会加速对该键的查询。然而，对于其他查询可能没有帮助，即便是查询包含了被索引的键。例如，下面的查询就不会从先前建立索引中获得任何的性能提升。

```
> db.people.find({"date" : date1}).sort({"date" : 1, "username" : 1})
```

服务器必须“查找整本书”找到想要的日期。这个过程称作*表扫描*，就是在没有索引的书中找内容，要从第一页开始，从前翻到后。通常来说，要尽量避免让服务器做表扫描，因为当集合很大时会非常慢。

实践证明，一定要创建查询中用到的所有键的索引。例如，对于上面的查询，应该建立日期和用户名的索引：

```
> db.ensureIndex({"date" : 1, "username" : 1})
```

传递给 `ensureIndex` 的文档其形式与传递给 `sort` 的文档形式一样：一组值为 1 或者 -1 的键，表示索引创建的方向。若索引只有一个键，则方向无关紧要。单键索引有点像一个按照字母顺序组织的书籍索引：无论从 A 到 Z，还是从 Z 到 A，显然都要从 M 开始查找。

若是有多个键，就得考虑索引的方向问题了。例如有如下的用户集合：

```
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
```

比如以 `{"username" : 1, "age" : -1}` 这种方式创建索引。MongoDB 会按如下方式组织用户：

```
{ "_id" : ..., "username" : "jacob", "age" : 17, "user_id" : 8 }
{ "_id" : ..., "username" : "joe", "age" : 36, "user_id" : 4 }
{ "_id" : ..., "username" : "joe", "age" : 27, "user_id" : 7 }
{ "_id" : ..., "username" : "john", "age" : 36, "user_id" : 2 }
{ "_id" : ..., "username" : "john", "age" : 18, "user_id" : 3 }
{ "_id" : ..., "username" : "john", "age" : 7, "user_id" : 5 }
{ "_id" : ..., "username" : "sally", "age" : 52, "user_id" : 9 }
```

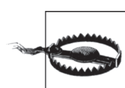
```
{ "_id" : ..., "username" : "simon", "age" : 59, "user_id" : 10 }
{ "_id" : ..., "username" : "simon", "age" : 3, "user_id" : 6 }
{ "_id" : ..., "username" : "smith", "age" : 48, "user_id" : 0 }
{ "_id" : ..., "username" : "smith", "age" : 30, "user_id" : 1 }
```

用户名严格地按照字母升序排列，同名的组按照年龄降序排列。这对 {"username" : 1, "age" :-1} 这样的排序做了优化，但是对 {"username" : 1, "age" : 1} 就不那么有效了。要是想对其优化的话，则需建立 {"username" : 1, "age" : 1} 的索引以便按照年龄升序组织。

对用户名和年龄的索引同样能加快对用户名的查询。一般来说，如果索引包含 N 个键，则对于前几个键的查询都会有帮助。比如有个索引 {"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}，实际上是有了 {"a" : 1}、{"a" : 1, "b" : 1}、{"a" : 1, "b" : 1, "c" :1} 等的索引。但是使用 {"b" : 1}、{"a" : 1, "c" : 1} 等索引的查询则不会被优化，只有使用索引前部的查询才能使用该索引。

MongoDB 的查询优化器会重排查询项的顺序，以便利用索引：比如查询 {"x" : "foo", "y" : "bar"} 的时候，已经有了 {"y" : 1, "x" : 1} 的索引，MongoDB 会自己找到并利用它。

创建索引的缺点就是每次插入、更新和删除时都会产生额外的开销。这是因为数据库不但需要执行这些操作，还要将这些操作在集合的索引中标记。因此，要尽可能少创建索引。每个集合默认的最大索引个数为 64 个，能够应付绝大多数情况了。



一定不要索引每一个键。这会导致插入非常慢，还会占用很多空间，并且很可能对查询速度提升不大。仔细考虑到底要做什么样的查询，什么样的索引适合这样的查询，通过 explain 和 hint 工具确保服务器使用了业已建立的索引，这两个工具会在下一节详细介绍。

有些时候，最有效的方法居然是不使用索引。一般说来，要是查询要返回集合中一半以上的结果，用表扫描会比几乎每条文档都查索引要高效一些。所以，查询是否存在某个键，或者检查某个布尔类型的值为真还是为假，真的没有用索引的必要。

5.1.1　扩展索引

假设我们有个集合，保存了用户的状态消息。现在想要查询用户和日期，取出某一用户最近的状态。以我们目前所学，我们会像下面这样创建一个索引：

```
> db.status.ensureIndex({user : 1, date : -1})
```

这会使对用户和日期的查询非常快，但是并不是最好的方式。

再想想书籍的索引。有一组文档按照用户名（升序）排序，尔后按着日期（降序）排序，所以会是这种情形：

```
User 123 on March 13, 2010
User 123 on March 12, 2010
User 123 on March 11, 2010
User 123 on March 5, 2010
User 123 on March 4, 2010
User 124 on March 12, 2010
User 124 on March 11, 2010
...
```

这点数据看着还行，但是应用会有数百万的用户，每人每天有数十条状态更新。若是每条用户状态的索引值占用类似一页纸的磁盘空间，那么对于每次“最新状态”的查询，数据库都会将不同的页载入内存。若是站点太热门，内存放不下所有的索引，就会非常非常慢。

要是改变索引的顺序，变成 {date : -1, user : 1}，则数据库可以将最后几天的索引保存在内存中，可以有效减少内存交换，这样查询任何用户的最新状态都会快很多。

所以，建立索引时要考虑如下问题。

(1) 会做什么样的查询？其中哪些键需要索引？
(2) 每个键的索引方向是怎样的？
(3) 如何应对扩展？有没有种不同的键的排列可以使常用数据更多地保留在内存中？

要是能回答这些问题，说明你已经做好了索引的准备了。

5.1.2　索引内嵌文档中的键

为内嵌文档的键建立索引和为普通的键创建索引没有什么区别。例如，要想按日期搜索博客文章的评论，可以在由内嵌的 "comments" 文档组成的数组中对 "date" 键创建索引：

```
> db.blog.ensureIndex({"comments.date" : 1})
```

对内嵌文档的键索引与普通键索引并无差异，两者也可以联合组成复合索引。

5.1.3　为排序创建索引

随着集合的增长，需要针对查询中大量的排序做索引。如果对没有索引的键调用 sort，MongoDB 需要将所有数据提取到内存来排序。因此，可以做无索引排序是有个上限的，那就是不可能在内存里面做 T 级别数据的排序。一旦集合大到不能在内存中排序，MongoDB 就会报错。

按照排序来索引以便让 MongoDB 按照顺序提取数据，这样就能排序大规模数据，而不必担心用光内存。

5.1.4 索引名称

集合中的每个索引都有一个字符串类型的名字，来唯一标识索引，服务器通过这个名字来删除或者操作索引。默认情况下，索引名类似 keyname1_dir1_keyname2_dir2_..._keynameN_dirN 这种形式，其中 keynameX 代表索引的键，dirX 代表索引的方向（1 或者 -1）。要是索引的键特别多，这样命名就略显愚笨了，不过还好可以通过 ensureIndex 的选项来指定自定义的名字：

```
> db.foo.ensureIndex({"a" : 1, "b" : 1, "c" : 1, ..., "z" : 1}, {"name"
                      : "alphabet"})
```

索引名有字符个数的限制，所以特别复杂的索引在创建时一定要使用自定义的名字。可以用 getLastError 来检查索引是否成功创建了或者未成功创建的原因。

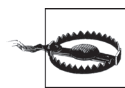
5.2 唯一索引

唯一索引可以确保集合的每一个文档的指定键都有唯一值。例如，如果想保证文档的 "username" 键都有不一样的值，创建一个唯一索引就好了：

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true})
```

一定要牢记默认情况下，insert 并不检查文档是否插入过了。所以，为了避免插入的文档中包含与唯一键重复的值，可能要用安全插入才能满足要求。这样，在插入这样的文档时会看到存在重复键错误的提示。

可能我们最熟悉的唯一索引就是对 "_id" 的索引了，这个索引是在创建普通集合时一同创建的。这个索引和普通唯一索引只有一点不同，就是不能删除。



如果没有对应的键，索引会将其作为 null 存储。所以，如果对某个键建立了唯一索引，但插入了多个缺少该索引键的文档，则由于文档包含 null 值而导致插入失败。

5.2.1 消除重复

当为已有的集合创建索引，可能有些值已经有重复了。若是真的发生这种情况，那么索引的创建就是失败。有些时候，可能希望将所有包含重复值的文档都删掉。dropDups 选项就可以保留发现的第一个文档，而删除接下来的有重复值的文档：

```
> db.people.ensureIndex({"username" : 1}, {"unique" : true, "dropDups"
                                            : true})
```

这种做法多少有点鲁莽，如果数据很重要的话，还是写个脚本做个预处理比较稳妥。

5.2.2 复合唯一索引

创建复合唯一索引的时候，单个键的值可以相同，只要所有键的值组合起来不同就好。

GirdFS 是 MongoDB 中存储大文件的标准方式（详见第 7 章），其中就用到了复合唯一索引。存储文件内容的集合有一个复合唯一索引 `{files_id : 1, n : 1}`，看起来就像：

```
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 2}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 3}
{files_id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 4}
```

注意，所有 `"files_id"` 的值都相同，但是 `"n"` 的值不同。若是试图再次插入 `{files_ id : ObjectId("4b23c3ca7525f35f94b60a2d"), n : 1}`，则数据库会提示存在重复键的错误。

5.3 使用explain和hint

`explain` 是一个非常有用的工具，会帮助你获得查询方面诸多有用的信息。只要对游标调用该方法，就可以得到查询细节。`explain` 会返回一个文档，而不是游标本身，这是与多数游标方法不同之处。

```
> db.foo.find().explain()
```

`explain` 会返回查询使用的索引情况（如果有的话），耗时及扫描文档数的统计信息。

例如，索引 `{"username" : 1}` 对单个键的查询非常有帮助，但是多数查询要复杂得多。比如，要做如下查询并排序：

```
> db.people.find({"age" : 18}).sort({"username" : 1})
```

这时就搞不太准数据库到底用没用已经创建的索引，或者到底效率如何。使用 `explain` 就会得到当前查询所使用的索引，消耗了多少时间，以及数据库需要扫描多少文档才能得到结果。

对于一个只有 64 个文档，没有索引（`"_id"` 索引除外）的数据库，做一次最简单的查询（`{}`），`explain` 的输出类似下面这样：

```
> db.people.find().explain()
{
    "cursor" : "BasicCursor",
    "indexBounds" : [ ],
    "nscanned" : 64,
    "nscannedObjects" : 64,
    "n" : 64,
    "millis" : 0,
    "allPlans" : [
        {
            "cursor" : "BasicCursor",
            "indexBounds" : [ ]
        }
    ]
}
```

结果中的要点如下。

`"cursor" : "BasicCursor"`

这说明查询没有使用索引（并不意外，因为没有查询条件）。一会儿会看到有索引的情形。

`"nscanned" : 64`

这个数字代表数据库查找了多少个文档。大家都想让这个数字尽可能地接近返回结果的数量。

`"n" : 64`

这个代表返回文档的数量。这个例子非常完美，因为扫描的文档数量和返回的文档数量完全一致。当然，这是由于返回了整个集合，否则的话很难做到。

`"millis" : 0`

这个毫秒数表示数据库执行查询的时间。0 是非常理想的成绩。

假设现在有一个基于 `"age"` 键的索引，现在要查找 20 多岁的用户。对这个查询使用 `explain` 会是这样的：

```
> db.c.find({age : {$gt : 20, $lt : 30}}).explain()
{
    "cursor" : "BtreeCursor age_1",
    "indexBounds" : [
        [
            {
                "age" : 20
            },
            {
                "age" : 30
            }
        ]
```

```
    ],
    "nscanned" : 14,
    "nscannedObjects" : 12,
    "n" : 12,
    "millis" : 1,
    "allPlans" : [
        {
            "cursor" : "BtreeCursor age_1",
            "indexBounds" : [
                [
                    {
                        "age" : 20
                    },
                    {
                        "age" : 30
                    }
                ]
            ]
        }
    ]
}
```

因为有了索引，和上个例子有些不一样，所以 explain 的几个输出键值发生了改变。

`"cursor" : "BtreeCursor age_1"`

查询不像先前一样使用了 BasicCursor。索引存储在 B 树的结构中，所以当使用索引查询，就会使用叫做 BtreeCursor 类型的游标。

这个值也标识了使用的索引名 age_1。通过这个名字，可以查询 *system.indexes* 集合来获取关于这个索引更进一步的信息（例如，是否是唯一索引，都包含哪些键）：

```
> db.system.indexes.find({"ns" : "test.c", "name" : "age_1"})
{
    "_id" : ObjectId("4c0d211478b4eaaf7fb28565"),
    "ns" : "test.c",
    "key" : {
        "age" : 1
    },
    "name" : "age_1"
}
```

`"allPlans" : [ ... ]`

这个键列举了所有 MongoDB 考虑的查询方案。当然这里的选择还是比较显然的，因为已经有了 "age" 索引，恰恰也是要查找 "age"。要是索引相互重叠，查询也很复杂，"allPlans" 就会包含所有可能用到的尝试。

看一个稍微复杂一点的例子。假设已经有 {"username" : 1, "age" : 1} 和 {"age" : 1, "username" : 1} 的索引了，现在要查询用户名和年龄，会怎么样

呢？要看具体的查询了：

```
> db.c.find({age : {$gt : 10}, username : "sally"}).explain()
{
        "cursor" : "BtreeCursor username_1_age_1",
        "indexBounds" : [
                [
                        {
                                "username" : "sally",
                                "age" : 10
                        },
                        {
                                "username" : "sally",
                                "age" : 1.7976931348623157e+308
                        }
                ]
        ],
        "nscanned" : 13,
        "nscannedObjects" : 13,
        "n" : 13,
        "millis" : 5,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor username_1_age_1",
                        "indexBounds" : [
                                [
                                        {
                                                "username" : "sally",
                                                "age" : 10
                                        },
                                        {
                                                "username" : "sally",
                                                "age" :
                                                1.7976931348623157e+308
                                        }
                                ]
                        ]
                }
        ],
        "oldPlan" : {
                "cursor" : "BtreeCursor username_1_age_1",
                "indexBounds" : [
                        [
                                {
                                        "username" : "sally",
                                        "age" : 10
                                },
                                {
                                        "username" : "sally",
                                        "age" : 1.7976931348623157e+308
                                }
                        ]
                ]
        }
}
```

这里的查询要求精确匹配用户名和年龄范围，所以数据库使用了{"username" : 1, "age" : 1}索引，而自己调换了查询项的顺序。另一方面，如果查找严格匹配的年龄和名字范围，MongoDB 就会使用别的索引：

```
> db.c.find({"age" : 14, "username" : /.*/}).explain()
{
        "cursor" : "BtreeCursor age_1_username_1 multi",
        "indexBounds" : [
                [
                        {
                                "age" : 14,
                                "username" : ""
                        },
                        {
                                "age" : 14,
                                "username" : {

                                }
                        }
                ],
                [
                        {
                                "age" : 14,
                                "username" : /.*/
                        },
                        {
                                "age" : 14,
                                "username" : /.*/
                        }
                ]
        ],
        "nscanned" : 2,
        "nscannedObjects" : 2,
        "n" : 2,
        "millis" : 2,
        "allPlans" : [
                {
                        "cursor" : "BtreeCursor age_1_username_1 multi",
                        "indexBounds" : [
                                [
                                        {
                                                "age" : 14,
                                                "username" : ""
                                        },
                                        {
                                                "age" : 14,
                                                "username" : {

                                                }
                                        }
                                ],
                                [
                                        {
```

```
                                        "age" : 14,
                                        "username" : /.*/
                                },
                                {
                                        "age" : 14,
                                        "username" : /.*/
                                }
                        ]
                ]
        }
    ]
}
```

如果发现 MongoDB 用了非预期的索引，可以用 `hint` 强制使用某个索引。例如，希望 MongoDB 在上一个例子中使用 `{"username" : 1, "age" : 1}` 索引，则需要：

```
> db.c.find({"age" : 14, "username" : /.*/}).hint({"username" : 1,
              "age" : 1})
```

多数情况下这种指定都没什么必要。MongoDB 的查询优化器非常智能，会替你选择该用哪个索引。初次做某个查询时，查询优化器会同时尝试各种查询方案。最先完成的被确定使用，其他的则终止掉。查询方案被记录下来，以备日后应对相同键的查询。查询优化器定期重试其他方案，以防因为添加新的数据后，之前的方案不再是最优的了。只要关心给查询优化器建立可以选择的索引就可以了。

5.4 索引管理

索引的元信息存储在每个数据库的 *system.indexes* 集合中。这是一个保留集合[1]，不能对其插入或者删除文档。操作只能通过 `ensureIndex` 或者 `dropIndexes` 进行。

system.indexes 集合包含每个索引的详细信息，同时 *system.namespaces* 集合也含有索引的名字。如果查看这个集合，会发现每个集合至少有两个文档与之对应，一个对应集合本身，一个对应集合包含的索引。对于只有标准的 `"_id"` 索引的集合，*system.namespaces* 应该类似这样：

```
{ "name" : "test.foo" }
{ "name" : "test.foo.$_id_"}
```

如果存在关于名字和年龄的复合索引，*system.namespaces* 则会增加一条文档：

```
{ "name" : "test.foo.$name_1_age_1" }
```

第 2 章讲到集合名的长度不得超过 121 字节。这个有点诡异的数字是因为 `"_id"` 索引的命名空间需要额外的 6 字节（`".$_id_"`），于是这个索引名的长度就是略显有意义的 127 字节了。

译注1：遍历数据库中的所有集合时要格外小心，因为通常我们并不想对这个集合执行操作。

一定要记住集合名和索引名加起来不能超过 127 字节。若是达到了命名空间的长度限制或是有很长的索引名，则需要自定义索引名，以避免过长。一般来说让数据库、集合、键的名字长度适中要比改名来得容易一些。

修改索引

随着你的应用和你一起慢慢变老，你会发现数据或者查询已经发生了改变，原来的索引也不那么好用了。不过使用 ensureIndex 随时可以向现有集合添加新的索引：

```
> db.people.ensureIndex({"username" : 1}, {"background" : true})
```

建立索引既耗时也费力，还需要消耗很多资源。使用 {"background" : true} 选项可以使这个过程在后台完成，同时正常处理请求。要是不包括 background 这个选项，数据库会阻塞建立索引期间的所有请求。

阻塞的做法会让索引建立得更快，同时也意味着应用在此期间不能应答。即便在后台进行也会对正常操作有些影响，所以最好选在无关紧要的时刻。后台创建索引也会增加些负载，好在不会让服务器停机。

为已有文档创建索引比先创建索引再插入所有文档要稍快一点。当然，要是集合的数据从无到有，事先创建一个索引也未尝不可。

要是索引没用了，便可以用 dropIndexes 加上索引名将其删除。通常，要查一下 system.indexes 集合来找出索引名，因为即便是自动生成的名字也会因为驱动程序不同而不同。

```
> db.runCommand({"dropIndexes" : "foo", "index" : "alphabet"})
```

要删除所有索引，可以将 index 的值赋为 *：

```
> db.runCommand({"dropIndexes" : "foo", "index" : "*"})
```

另外一种删除索引的方式就是删除集合。这也会删除 _id 索引（还有集合的所有文档）。删除集合的所有文档（用 remove 的方式）并不影响索引，当有新文档插入时还会再生的。

5.5 地理空间索引

还有一种查询变得越来越流行（尤其是随着移动设备的出现）：找到离当前位置最近的 N 个场所。MongoDB 为坐标平面查询提供了专门的索引，称作*地理空间索引*。

假设要找到给定经纬度坐标周围最近的咖啡馆，就需要创建一个专门的索引来提高这种查询的效率，这是因为这种查询需要搜索两个维度。地理空间索引同样可以由

ensureIndex 来创建，只不过参数不是 1 或者 -1，而是 "2d"：

```
> db.map.ensureIndex({"gps" : "2d"})
```

"gps" 键的值必须是某种形式的一对值：一个包含两个元素的数组或是包含两个键的内嵌文档。下面这些都是有效的：

```
{ "gps" : [ 0, 100 ] }
{ "gps" : { "x" : -30, "y" : 30 } }
{ "gps" : { "latitude" : -180, "longitude" : 180 } }
```

至于键名可以随意，例如 {"gps" : {"foo" : 0, "bar" : 1}} 也是可以的。

默认情况下，地理空间索引假设值的范围是 -180~180（对经纬度来说很方便）。要是想用其他值，可以通过 ensureIndex 的选项来指定最大最小值：

```
> db.star.trek.ensureIndex({"light-years" : "2d"}, {"min" : -1000,
                                                    "max" : 1000})
```

这样就创建了一个 2000 光年见方的空间索引。

地理空间查询以两种方式进行，即普通查询（用 find）或者使用数据库命令。find 的方式与一般的查询差别不大，只不过用了 "$near"。需要两个目标值的数组作为参数：

```
> db.map.find({"gps" : {"$near" : [40, -73]}})
```

这会按照离点（40，-73）由近及远的方式将 map 集合的所有文档都返回。在没有指定 limit 的值时，默认是 100 个文档。要是不需要这么多结果，就应该设置一个少点的值以节约资源。例如，下面的例子将返回离（40，-73）最近的 10 个文档。

```
> db.map.find({"gps" : {"$near" : [40, -73]}}).limit(10)
```

也可以用 geoNear 来完成相同的操作：

```
> db.runCommand({geoNear : "map", near : [40, -73], num : 10});
```

geoNear 还会返回每个文档到查询点的距离。这个距离是以你插入的数据为单位的，如果按照经纬度的角度插入，则距离就是经纬度。find 与 "$near" 的组合不会给出距离，但若是结果大于 4 MB，这是唯一的选择。

MongoDB 不但能找到靠近一个点的文档，还能找到指定形状内的文档。做法就是将原来的 "$near" 换成 "$within"。"$within" 获取数量不断增加的形状作为参数。若查看地理空间索引的联机帮助文档（http://www.mongodb.org/display/DOCS/Geospatial+ Indexing），可以找到最新的形状列表。撰写本书时，有两个选项：你可以查询矩形和圆形内的所有点。

对于矩形，使用 "$box" 选项：

```
> db.map.find({"gps" : {"$within" : {"$box" : [[10, 20], [15, 30]]}}})
```

"$box" 参数是两个元素的数组，第一个元素指定了左下角的坐标，第二个指定右上角的坐标。

同样，也可以用 "$center" 来找到圆形内部的所有点，只不过参数变成了圆心和半径：

```
> db.map.find({"gps" : {"$within" : {"$center" : [[12, 25], 5]}}})
```

5.5.1　复合地理空间索引

应用经常要找的东西往往不只是一个地点。例如，用户要找出周围所有的咖啡店或者披萨店。将地理空间索引与普通索引组合起来就可以满足这种需求。例如，要查询 "location" 和 "desc"，就可以这样创建索引：

```
> db.ensureIndex({"location" : "2d", "desc" : 1})
```

然后就能很快找到最近的咖啡馆了：

```
> db.map.find({"location" : {"$near" : [-70, 30]}, "desc" :
"coffeeshop"}).limit(1)
{
    "_id" : ObjectId("4c0d1348928a815a720a0000"),
    "name" : "Mud",
    "location" : [x, y],
    "desc" : ["coffee", "coffeeshop", "muffins", "espresso"]
}
```

注意，创建一个关键词数组对于用户自定义查找很有帮助。

5.5.2　地球不是二维平面

MongoDB 的地理空间索引假设索引内容是在一个平面上的。这就意味着对于球体，比如地球，它并不是十分精确，尤其是在极地区域。具体来说，两条经线之间纬线的长度在赤道和在育空地区[1]是大不一样的，后者要短得多。可以用不同的投影手段将地球映射到二维平面，当然它们的精度和相应的复杂度各不相同。

译注1：加拿大最西部的联邦领地。

第 6 章
聚合

MongoDB 除了基本的查询功能，还提供了很多强大的聚合工具，其中简单的可计算集合中的文档个数，复杂的可利用 MapReduce 做复杂数据分析。

6.1 count

`count` 是最简单的聚合工具，返回集合中的文档数量：

```
> db.foo.count()
0
> db.foo.insert({"x" : 1})
> db.foo.count()
1
```

不论集合有多大，都会很快返回总的文档数量。

也可以传递查询，Mongo 则会计算查询结果的数量：

```
> db.foo.insert({"x" : 2})
> db.foo.count()
2
> db.foo.count({"x" : 1})
1
```

对分页来说有个总数非常必要："共 439 个，目前显示 0-10。"然而增加查询条件会使得 `count` 变慢。

6.2 distinct

`distinct` 用来找出给定键的所有不同的值。使用时必须指定集合和键。

```
> db.runCommand({"distinct" : "people", "key" : "age"})
```

例如，假设有如下文档：

```
{"name" : "Ada", "age" : 20}
{"name" : "Fred", "age" : 35}
{"name" : "Susan", "age" : 60}
{"name" : "Andy", "age" : 35}
```

如果对 "age" 键使用 distinct，会获得所有不同的年龄：

```
> db.runCommand({"distinct" : "people", "key" : "age"})
{"values" : [20, 35, 60], "ok" : 1}
```

这里还有一个常见问题：有没有种方法获得集合里面所有不同的键呢？起码没有内置的，不过可以用 MapReduce 自己写一个（后面会讲到）。

6.3 group

group 做的聚合稍复杂一些。先选定分组所依据的键，而后 MongoDB 就会将集合依据选定键值的不同分成若干组。然后可以通过聚合每一组内的文档，产生一个结果文档。



如果你熟悉 SQL，那么这个 group 和 SQL 中的 GROUP BY 差不多。

假设现在有个站点要跟踪股票价格。从上午 10 点到下午 4 点每隔几分钟就更新一下某只股票的价格，并保存在 MongoDB 中。现在报表程序要获得近 30 天的收盘价。用 group 就可以很容易地办到。

股价的集合中包含数以千计的如下形式的文档：

```
{"day" : "2010/10/03","time" : "10/3/2010 03:57:01 GMT-400","price" : 4.23}
{"day" : "2010/10/04","time" : "10/4/2010 11:28:39 GMT-400","price" : 4.27}
{"day" : "2010/10/03","time" : "10/3/2010 05:00:23 GMT-400","price" : 4.10}
{"day" : "2010/10/06","time" : "10/6/2010 05:27:58 GMT-400","price" : 4.30}
{"day" : "2010/10/04","time" : "10/4/2010 08:34:50 GMT-400","price" : 4.01}
```



记着由于精度的问题，绝不要将金额以浮点数的方式存储，这里这么做只是为了简化例子。

想获得的结果就是每天最后的价格列表，就像这样：

```
[
    {"time" : "10/3/2010 05:00:23 GMT-400", "price" : 4.10},
```

```
    {"time" : "10/4/2010 11:28:39 GMT-400", "price" : 4.27},
    {"time" : "10/6/2010 05:27:58 GMT-400", "price" : 4.30}
]
```

先把集合按照天分组，然后在每一组里取包含最新时间戳的文档，将其放置到结果中就完成了。整个过程就像这样：

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }
... }}})
```

分解开来看看。

`"ns" : "stocks"`

指定要进行分组的集合。

`"key" : "day"`

指定文档分组依据的键。这里就是 `"day"` 键。所有 `"day"` 值相同的文档被划分到一组。

`"initial" : {"time" : 0}`

每一组 reduce 函数调用的初始时间，会作为初始文档传递给后续过程。每一组的所有成员都会使用这个累加器，所以改变会保留住。

`"$reduce" : function(doc, prev) { ... }`

每个文档都对应一次这个调用。系统会传递两个参数：当前文档和累加器文档（本组当前的结果）。本例中，想让 reduce 函数比较当前文档的时间和累加器的时间。如果当前文档的时间更近，则将累加器的日期和价格替换成当前文档的值。别忘了，每一组都有一个独立的累加器，所以不必担心不同的日期使用同一个累加器。

在问题一开始的描述中，就提到只要最近 30 天的股价。然而，这里迭代了整个集合。这就是为什么要添加 `"condition"`，因为这样就可以只处理满足条件的文档了。

```
> db.runCommand({"group" : {
... "ns" : "stocks",
... "key" : "day",
... "initial" : {"time" : 0},
... "$reduce" : function(doc, prev) {
```

```
...     if (doc.time > prev.time) {
...         prev.price = doc.price;
...         prev.time = doc.time;
...     }},
... "condition" : {"day" : {"$gt" : "2010/09/30"}}
... }})
```



有些参考资料提及 "cond" 键或者 "q" 键，其实和 "condition" 键是完全一样的（就是看上去短点）。

最后就会返回由 30 个文档组成的数组，每个组一个文档。每组还有分组依据的键（这里就是 "day" : *string*）以及这组最终的 prev 值。如果有的文档没有依据的键，就都会被分到一组，相应的部分就会使用 "day" : null" 这样的形式。在 "condition" 中加入 "day" : {"$exists" : true} 就可以去掉这组。group 命令还会返回使用的文档总数和 "key" 有多少个不同的值：

```
> db.runCommand({"group" : {...}})
{
    "retval" :
        [
            {
                "day" : "2010/10/04",
                "time" : "Mon Oct 04 2010 11:28:39 GMT-0400 (EST)"
                "price" : 4.27
            },
            ...
        ],
    "count" : 734,
    "keys" : 30,
    "ok" : 1
}
```

这里每组的 "price" 都是显式设置的，"time" 先由初始化器设置，然后也是主动更新。"day" 是默认被加进去的，因为分组依据的键默认被加入到每个 "retval" 内嵌文档中。要是不想返回这个键，可以用完成器把累加器文档变成任意形态，甚至变换成非文档（例如数字或字符串）。

6.3.1 使用完成器

完成器（finalizer）用以精简从数据库传到用户的数据，这个步骤非常重要，因为 group 命令的输出一定要能放在单个数据库响应中。为进一步说明，这里举个博客的例子，其中每篇文章都有多个标签（tag）。现在要找出每天最热点的标签。可以（再一次）按天分组，为每一个标签计数。就像下面这样：

```
> db.posts.group({
```

```
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     }
... }})
```

结果会是这样：

```
[
    {"day" : "2010/01/12", "tags" : {"nosql" : 4, "winter" : 10, "sledding" :2}},
    {"day" : "2010/01/13", "tags" : {"soda" : 5, "php" : 2}},
    {"day" : "2010/01/14", "tags" : {"python" : 6, "winter" : 4, "nosql": 15}}
]
```

接着可以在客户端找出 `"tags"` 文档中值最大的标签。然而，向客户端发送每天所有的标签文档需要许多额外的开销：每天所有的键 / 值对都传送，而我们仅仅需要单个字符串。这也就是 group 有一个 `"finalize"` 键的原因。`"finalize"` 附带一个函数，在每组结果传递到客户端之前被调用一次。可以用其修剪结果中的“残枝败叶”：

```
> db.runCommand({"group" : {
... "ns" : "posts",
... "key" : {"tags" : true},
... "initial" : {"tags" : {}},
... "$reduce" : function(doc, prev) {
...     for (i in doc.tags) {
...         if (doc.tags[i] in prev.tags) {
...             prev.tags[doc.tags[i]]++;
...         } else {
...             prev.tags[doc.tags[i]] = 1;
...         }
...     },
... "finalize" : function(prev) {
...     var mostPopular = 0;
...     for (i in prev.tags) {
...         if (prev.tags[i] > mostPopular) {
...             prev.tag = i;
...             mostPopular = prev.tags[i];
...         }
...     }
...     delete prev.tags
... }}})
```

现在好了，仅返回希望的结果。服务器返回：

```
[
    {"day" : "2010/01/12", "tag" : "winter"},
```

```
    {"day" : "2010/01/13", "tag" : "soda"},
    {"day" : "2010/01/14", "tag" : "nosql"}
]
```

finalize 能修改传递的参数也能返回新值。

6.3.2 将函数做为键使用

有些时候分组所依据的条件非常复杂，不仅是一个键。比如要使用 group 计算每个类别有多少篇博客文章（每篇文章只属于一个类别）。由于有很多作者，给文章分类时可能不规律地用了大小写。所以，如果要是按类别名来分组，最后“MongoDB”和“mongodb”就是两个完全不同的组。为了消除这种大小写的影响，就要定义一个函数来确定文档分组所依据的键。

定义分组函数就要用到 $keyf 键（注意不是 "key"）。就像这样：

```
> db.posts.group({"ns" : "posts",
... "$keyf" : function(x) { return x.category.toLowerCase(); },
... "initializer" : ... })
```

有了 "$keyf" 就能依据各种复杂的条件进行分组了。

6.4 MapReduce

MapReduce 是聚合工具中的明星。count、distinct、group 能做的上述事情 MapReduce 都能做。它是一个可以轻松并行化到多个服务器的聚合方法。它会拆分问题，再将各个部分发送到不同的机器上，让每台机器都完成一部分。当所有机器都完成的时候，再把结果汇集起来形成最终完整的结果。

MapReduce 需要几个步骤。最开始是映射（map），将操作映射到集合中的每个文档。这个操作要么“无作为”，要么“产生一些键和 X 个值”。然后就是中间环节，称作洗牌（shuffle），按照键分组，并将产生的键值组成列表放到对应的键中。化简（reduce）则把列表中的值化简成一个单值。这个值被返回，然后接着进行洗牌，直到每个键的列表只有一个值为止，这个值也就是最后结果。

使用 MapReduce 的代价就是速度：group 不是很快，MapReduce 更慢，绝不要用在“实时”环境中。要作为后台任务来运行 MapReduce，将创建一个保存结果的集合，可以对这个集合进行实时查询。

下面会多举几个 MapReduce 的例子，这个工具非常强大，但也有点复杂。

6.4.1　例1：找出集合中的所有键

用 MapReduce 来解决这个问题有点大材小用，不过还是一种了解其机制的不错的方式。要是已经知道 MapReduce 的原理，则直接跳到本节最后，看看 MongoDB 中用 MapReduce 的注意事项。

MongoDB 没有模式，所以并不知晓每个文档有多少个键。通常找到集合的所有键的最好方式就是用 MapReduce。在本例中，还会记录每个键都出现了多少次。内嵌文档中的键就不计算了，但给 map 函数做个简单修改就能实现这个功能了。

在映射环节，想得到文档中的每个键。map 函数使用函数 emit“返回”要处理的值。emit 会给 MapReduce 一个键（类似于前面 group 所使用的键）和一个值。这里用 emit 将文档某个键的计数（count）返回（{count : 1}）。我们想为每个键单独计数，所以为文档中的每一个键调用一次 emit。this 就是当前映射文档的引用：

```
> map = function() {
... for (var key in this) {
...     emit(key, {count : 1});
... }};
```

这样就有了许许多多 {count : 1} 文档，每一个都与集合中的一个键相关。这种由一个或多个 {count : 1} 文档组成的数组，会传递给 reduce 函数。reduce 函数有两个参数，一个是 key，也就是 emit 返回的第一个值，还有另外一个数组，由一个或者多个对应于键的 {count : 1} 文档组成。

```
> reduce = function(key, emits) {
... total = 0;
... for (var i in emits) {
...     total += emits[i].count;
... }
... return {"count" : total};
... }
```

reduce 一定要能被反复调用，不论是映射环节还是前一个简化环节。所以 reduce 返回的文档必须能作为 reduce 的第二个参数的一个元素。例如，x 键映射到了 3 个文档 {count : 1, id : 1}、{count : 1, id : 2} 和 {count :1, id : 3}，其中 id 键用于区别。MongoDB 可能这样调用 reduce：

```
> r1 = reduce("x", [{count : 1, id : 1}, {count : 1, id : 2}])
{count : 2}
> r2 = reduce("x", [{count : 1, id : 3}])
{count : 1}
> reduce("x", [r1, r2])
{count : 3}
```

不能认为第二个参数总是初始文档之一（这里便是 {count : 1}）或者有固定长

度。reduce 应该能处理 emit 文档和其他 reduce 结果的各种组合。

总之，MapReduce 函数类似这样：

```
> mr = db.runCommand({"mapreduce" : "foo", "map" : map, "reduce" :
reduce})
{
    "result" : "tmp.mr.mapreduce_1266787811_1",
    "timeMillis" : 12,
    "counts" : {
        "input" : 6
        "emit" : 14
        "output" : 5
    },
    "ok" : true
}
```

MapReduce 返回的文档包含很多与操作有关的元信息：

```
"result" : "tmp.mr.mapreduce_1266787811_1"
```

这是存放 MapReduce 结果的集合名。这是个临时集合，MapReduce 的连接关闭后自动就被删除了。本章稍后会讲如何指定一个好一点的名字和如何让集合具有持久性。

`"timeMillis" : 12`

操作花费的时间，单位是毫秒。

`"counts" : { ... }`

这个内嵌文档包含 3 个键。

`"input" : 6`

发送到 map 函数的文档个数。

`"emit" : 14`

在 map 函数中 emit 被调用的次数。

`"output" : 5`

结果集合中创建的文档数量。

"counts" 对调试非常有帮助。

对结果集合进行查询会发现原有集合的所有键及其计数：

```
> db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
```

```
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

每个键值变为一个 `"_id"`，最终化简步骤的结果变为 `"value"`。

6.4.2 例2：网页分类

假设有个网站，人们可以提交其他网页的链接，比如 reddit.com。提交者可以给这个链接做标签，表明主题，比如“politics”“geek”或者“icanhascheezburger”。可以用 MapReduce 找出哪个主题最为热门，热门与否由最近的投票决定。

首先，建立一个 `map` 函数，发出（`emit`）标签和一个基于流行度和新近程度的值。

```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;

        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

现在就化简同一个标签的所有值，形成这个标签的分数：

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

最终的集合包含每个标签的 URL 列表和表示该标签流行程度的分数。

6.4.3 MongoDB和MapReduce

前面两个例子只用到了 `mapreduce`、`map` 和 `reduce` 键。这 3 个键是必需的，但是 MapReduce 命令还有很多可选的键。

- `"finalize"`：函数

  将 `reduce` 的结果发送给这个键，这是处理过程的最后一步。

- `"keeptemp"`：布尔

连接关闭时临时结果集合是否保存。

- "output"：字符串
结果集合的名字。设定该项则隐含着 keeptemp : true。

- "query"：文档
会在发往 map 函数前，先用指定条件过滤文档。

- "sort"：文档
在发往 map 前先给文档排序（与 limit 一同使用非常有用）。

- "limit"：整数
发往 map 函数的文档数量的上限。

- "scope"：文档
JavaScript 代码中要用到的变量。

- "verbose"：布尔
是否产生更加详尽的服务器日志。

1. finalize函数

和 group 命令一样，MapReduce 也可以使用 finalize 函数作为参数。它会在最后 reduce 得到输出后执行，然后将结果存到临时集合中。

体积大点的结果对 MapReduce 来说还好，因为不像 group 那样有 4 MB 的限制。然而，无论如何信息还是要传递的，所以 finalize 就是一个计算平均数、裁剪数组、清除多余信息的恰当时机。

2. 保留结果集合

默认情况下，Mongo 会在执行 MapReduce 的时候创建一个临时集合，集合名是系统选的一个常人不太会用的名字，其中含有 mr，执行 MapReduce 的集合名，时间戳，数据库作业 ID，将这些用“.”连成一个字符串。结果产生形如“mr.stuff.18234210220.2”这样的名字。MongoDB 会在调用的连接关闭时自动销毁这个集合（也可以在用完之后手动删除）。如果想保留这个集合，就要指定 keeptemp 为 true。

如果要经常使用这个临时集合，没准想给它起个好点的名字。利用 out 选项（该选项接受字符串作为参数）就可以指定一个人类易懂的名字。如果用了 out 选项，就不必指定 keeptemp : true 了，因为已经隐含在其中了。即便起了一个非常好的名字，MongoDB 也会在 MapReduce 的中间过程使用自动生成的集合名。处理完成

后，自动更改成指定的名字，这个过程是原子的。也就是说，如果多次对同一个集合调用 MapReduce，也不会出现使用不完整集合的情况。

MapReduce 产生的集合就是一个普通的集合，在其上执行 MapReduce 一点问题也没有，或者在前一个 MapReduce 的结果上执行 MapReduce 也没有问题，如此往复直到无穷都没问题！

3. 对文档子集合执行MapReduce

有时候需要对集合的一部分执行 MapReduce。只需要在传给 map 函数前添加一个查询来过滤一下文档就好了。

每个传递给 map 函数的文档都要事先反序列化，从 BSON 转换成 JavaScript 对象，这个过程非常耗资源。要是事先能确定只对集合的一部分文档执行 MapReduce，增加一层过滤会极大地提高速度。过滤也无非就是用 "query"、"limit" 和 "sort" 键指定的。

"query" 键的值是一个查询文档。通常查询返回的结果就传递给了 map 函数。例如，有个应用程序做跟踪分析，需要上周的概要，只要使用如下命令对上周的文档执行 MapReduce 就好了：

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "query" : {"date" : {"$gt" : week_ago}}})
```

sort 选项一般和 limit 一同发挥重要作用。limit 也可以单独使用，用来截取一部分文档发送给 map 函数。

如果在上个例子中想分析最近 10 000 个页面视图（而不是最近一周的），则可以借助 limit 和 sort：

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "limit" : 10000, "sort" : {"date" : -1}})
```

query、limit、sort 可以随意组合，但要是没有 limit，sort 单独使用的用处不大。

4. 使用作用域

MapReduce 可以为 map、reduce、finalize 函数都采用一种代码类型。但多数语言里，可以指定传递代码的作用域。然而 MapReduce 会忽略这个作用域。它有其自己的作用域键 "scope"，如果想在 MapReduce 中使用客户端的值，则必须使用这个参数。可以用“变量名：值”这样的普通文档来设置该选项，然后在 map、reduce 和 finalize 函数中就能使用了。作用域在这些函数内部是不变的。

例如，在上一节的例子中，用 1/(new Date() - this.date) 计算了页面的新近程度。还可以将当前日期作为作用域的一部分传递进去：

```
> db.runCommand({"mapreduce" : "webpages", "map" : map, "reduce" : reduce,
                 "scope" : {now : new Date()}})
```

这样，在 map 函数中就能计算 1/(now - this.date) 了。

5. 获得更多的输出

还有个用于调试的详细输出选项。如果想看看 MapReduce 的运行过程，可以用 "verbose": true。

也可以用 print 把 map、reduce、finalize 过程中的信息输出到服务器日志上。

第 7 章

进阶指南

MongoDB 支持一些先前没有讨论的高级功能。要想成为高级用户，一定得看看本章，本章具体会涵盖下列主题。

- 通过数据库命令使用高级特性。
- 使用一种特殊的集合——固定大小的集合。
- 使用 GridFS 存储大文件。
- 利用 MongoDB 对服务端 JavaScript 的支持。
- 理解何为数据库引用，何时该使用。

7.1 数据库命令

前面的章节讲到了如何在 MongoDB 中创建、读取、更新、删除文档。除了这些基本操作，MongoDB 还支持大量的高级操作，这些操作都是用命令实现的。除了“创建、读取、更新和删除”，其他的功能都是作为命令实现的。

前面的章节已经涵盖了一些命令。例如，第 3 章讲了使用 `getLastError` 来查看一个更新对多少个文档起作用：

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

本节会深入研究命令究竟是什么，它们是如何实现的。还会介绍一些 MongoDB 中

最常用的命令。

7.1.1 命令的工作原理

可能大家都熟悉命令 drop：要在 shell 中删除一个集合，执行 db.test.drop()。在幕后，这个函数实际运行的是 drop 命令，可以用 runCommand 来达到完全一样的效果：

```
> db.runCommand({"drop" : "test"});
{
    "nIndexesWas" : 1,
    "msg" : "indexes dropped for collection",
    "ns" : "test.test",
    "ok" : true
}
```

命令的响应是作为结果的一个文档，包含了命令是否成功执行，还可能有些其他的命令输出的信息。命令的响应应该总是有 "ok" 这个键。如果 "ok" 的值是 true，代表命令成功执行，如果为 false，就代表命令执行出了某些问题。



1.5 及之前的版本，"ok" 的值是 1.0 或者 0.0，而不是 true 和 false。

如果 "ok" 为 false，还会有另外的一个叫 "errmsg" 的键。"errmsg" 的值为一个字符串，表示命令失败的原因。例如，如果对刚刚删除的集合再次运行 drop 命令：

```
> db.runCommand({"drop" : "test"});
{ "errmsg" : "ns not found", "ok" : false }
```

MongoDB 中的命令其实是作为一种特殊类型的查询来实现的，这些查询针对 $cmd 集合来执行。runCommand 仅仅是接受命令文档，执行等价查询，因此 drop 调用实际上是这样的：

```
db.$cmd.findone({"drop" : "test"});
```

当 MongoDB 服务器得到查询 $cmd 集合的请求时，会启动一套特殊的逻辑来处理，而不是交给普通的查询代码来执行。几乎所有 MongoDB 驱动程序都提供一个类似于 runCommand 的帮助方法来执行命令，但是如果有必要，总是可以使用一个简单查询的方式来运行命令。

访问有些命令需要有管理员权限，必须在 admin 数据库里面运行。如果在别的数据库里运行这样的命令，会得到“拒绝访问”的错误。

7.1.2 命令参考

本书写作时，MongoDB 支持超过 75 个命令[1]，而且今后会有更多命令的。要获得所有命令的最新列表，有两种方式。

- 在 shell 中运行 `db.listCommands()`，或者从驱动程序中运行等价的命令 `listCommands`。
- 浏览管理员接口 http://localhost:28017/_commands（关于管理员接口详见第 8 章）。

下面列举了 MongoDB 中最经常使用的命令，并给出了示例文档，以说明这些命令是如何表示的。

- `buildInfo`

  `{"buildInfo" : 1}`

  管理专用命令，返回 MongoDB 服务器的版本号和主机的操作系统。

- `collStats`

  `{"collStats" : collection}`

  返回指定集合的统计信息，包括数据大小、已分配的存储空间和索引的大小。

- `distinct`

  `{"distinct" : collection, "key": key, "query": query}`

  列出指定集合中满足查询条件的文档的指定键的所有不同值。

- `drop`

  `{"drop" : collection}`

  删除集合的所有数据。

- `dropDatabase`

  `{"dropDatabase" : 1}`

  删除当前数据库的所有数据。

- `dropIndexes`

  `{"dropIndexes" : collection, "index" : name}`

  删除集合里面名称为 `name` 的索引，如果名称为 `"*"`，则删除全部索引。

- `findAndModify`

  `findAndModify` 的用法详见第 3 章。

译注1：到2011年4月，已经有103个命令了。

- getLastError

`{"getLastError" : 1[, "w" : w[, "wtimeout" : timeout]]}`

查看对本集合执行的最后一次操作的错误信息或者其他状态信息。在 *w* 台服务器复制集合的最后操作之前，这个命令会阻塞（超时的毫秒数到了）。

- isMaster

`{"isMaster" : 1}`

检查本服务器是主服务器还是从服务器。

- listCommands

`{"listCommands" : 1}`

返回所有可以在服务器上运行的命令及相关信息。

- listDatabases

`{"listDatabases" : 1}`

管理专用命令，列出服务器上所有的数据库。

- ping

`{"ping" : 1}`

检查服务器链接是否正常。即便服务器上锁了，这条命令也会立刻返回。

- renameCollection

`{"renameCollection" : a, "to" : b}`

将集合 `a` 重命名为 `b`，其中 `a` 和 `b` 都必须是完整的集合命名空间（例如 `"foo.bar"` 表示 `foo` 数据库中的 `bar` 集合）。

- repairDatabase

`{"repairDatabase" : 1}`

修复并压缩当前数据库，这个操作可能非常耗时。详见 8.4.5 节。

- serverStatus

`{"serverStatus" : 1}`

返回这台服务器的管理统计信息。详见 8.2 节。

要记住，上面列举的只是一小部分命令。本书后面还会涉及一些，要查看完整的列表，只要运行 `listCommands` 就可以了。

7.2 固定集合

前面已经介绍了，MongoDB 如何动态建立普通集合，如何应对增长的数据自动调整大小。MongoDB 还支持另外一种集合——固定集合，要事先创建，而且大小固定（参见图 7-1）。固定大小的集合带来个有趣的问题：如何向一个满的固定集合插入数据呢？答案是固定集合很像环形队列，如果空间不足，最早的文档就会被删除，为新的文档腾出空间（参见图 7-2）。这意味着固定集合在新文档插入的时候自动淘汰最早的文档。

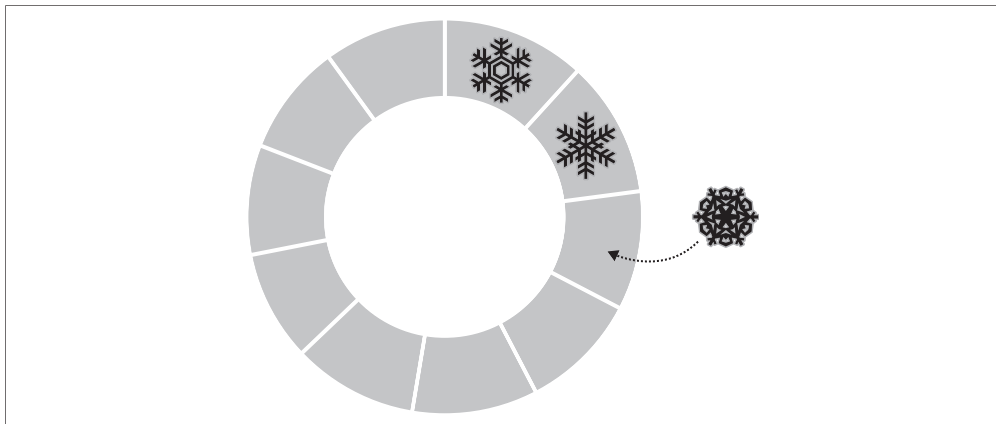


图 7-1：插入新文档到队尾

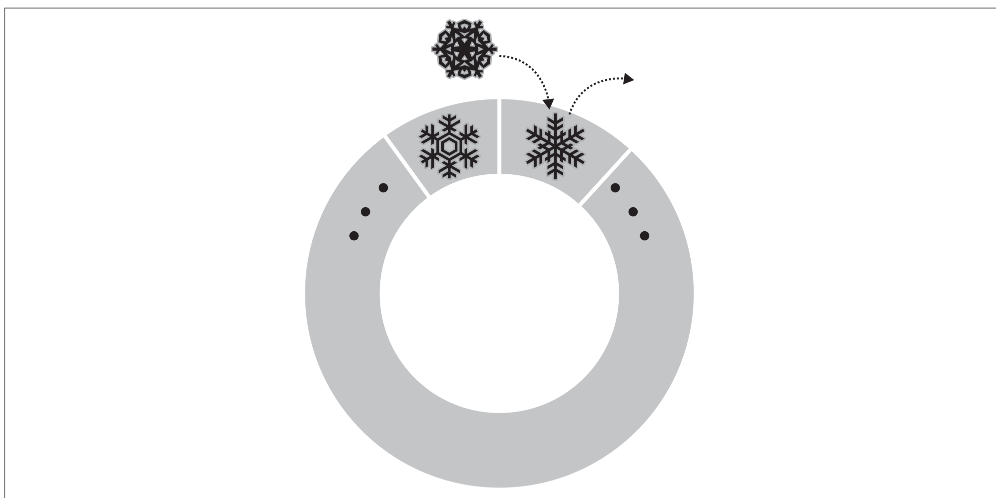


图 7-2：当队列满了，新的元素会将最早的元素替换掉

有些操作不适用于固定集合。不能删除文档（除了前面所说的自动淘汰），更新不得导致文档移动（通常更新意味着尺寸增大）。基于上述两点，可以保证在固定集合中的文档以插入的顺序存储，而且不必维护一个已删除的文档的释放空间列表。

固定集合和普通集合还有一个区别，就是在默认情况下固定集合没有索引，即便是 `"_id"` 上也没有索引。

7.2.1 属性及用法

固定集合的功能与限制合二为一，给我们带来了些有趣的特性。第一，对固定集合进行插入速度极快。做插入操作时，无需额外分配空间，服务器也不必查找空闲列表来放置文档。直接将文档插入集合的“末尾”就好了，如有必要就将旧的覆盖。默认情况下插入也无需更新索引，所以插入实际上是一个简单的 `memcpy`。

第二个有趣的属性就是按照插入顺序输出的查询速度极快。因为文档本身就是按照插入顺序存储的，按照这个顺序查询就是遍历一下，返回结果的顺序就是文档在磁盘上的顺序。默认情况下，对固定集合进行查找都会以插入顺序返回结果。

最后一个属性，固定集合能够在新数据插入时，自动淘汰最早的数据。插入快速、按照插入顺序查询也快速、自动淘汰，这几样组合起来使得固定集合特别适合像日志这种应用场景。事实上，MongoDB 中设计固定集合的目的就是用来存储内部的复制日志 oplog（关于复制和 oplog，详见第 9 章）。固定集合还有个很好的用法，就是缓存少量的文档。一般来说，固定集合适用于任何想要自动淘汰过期属性的场景，没有太多的操作限制。

7.2.2 创建固定集合

不像普通集合，固定集合必须要在使用前显式地创建。使用 `create` 命令创建。在 shell 中，可以使用 `createCollection` 来创建：

```
> db.createCollection("my_collection", {capped: true, size: 100000});
{ "ok" : true }
```

上面的命令创建了一个固定集合 `my_collection`，大小是 100 000 字节。`createCollection` 也有些别的选项。除了指定总的容量，还可以指定文档数量的上限：

```
> db.createCollection("my_collection", {capped: true, size: 100000,
                                        max: 100});
{ "ok" : true }
```



当指定文档数量上限时，必须同时指定大小。淘汰机制只有在容量还没有满时才会依据文档数量来工作。要是容量满了，淘汰机制则会依据容量来工作，就像别的固定集合一样。

还可以通过转换已有的普通集合的方式来创建固定集合。使用 convertToCapped 命令来完成这个操作。下面的例子中，会把 test 集合转换成大小为 10 000 字节的固定集合。

```
> db.runCommand({convertToCapped: "test", size: 10000});
{ "ok" : true }
```

7.2.3 自然排序

固定集合有种特殊的排序方式，叫做*自然排序*。自然顺序就是文档在磁盘上的顺序（见图 7-3）。

因为固定集合的文档总是按照插入的顺序存储的，自然顺序就是与此相同的。前面讲到过，在默认情况下，查询固定集合后就是按照插入顺序返回文档。也可以使用自然排序按照反向插入的顺序查询（见图 7-4）。

```
> db.my_collection.find().sort({"$natural": -1})
```

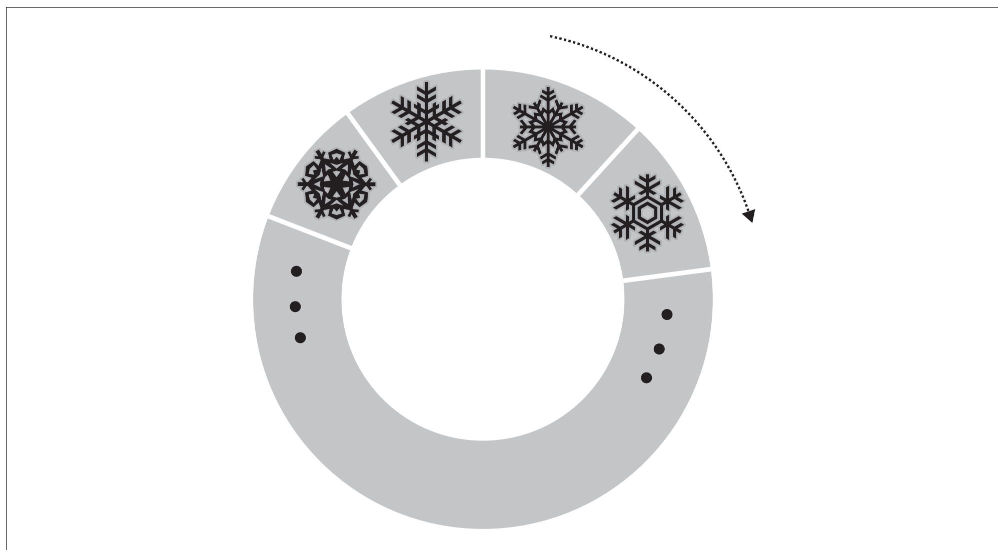


图 7-3：按照 {"$natural" : 1} 排序

使用 {"$natural": 1} 表示与默认顺序相同。非固定集合不能保证文档按照特定顺序存储，所以自然顺序的意义不大。

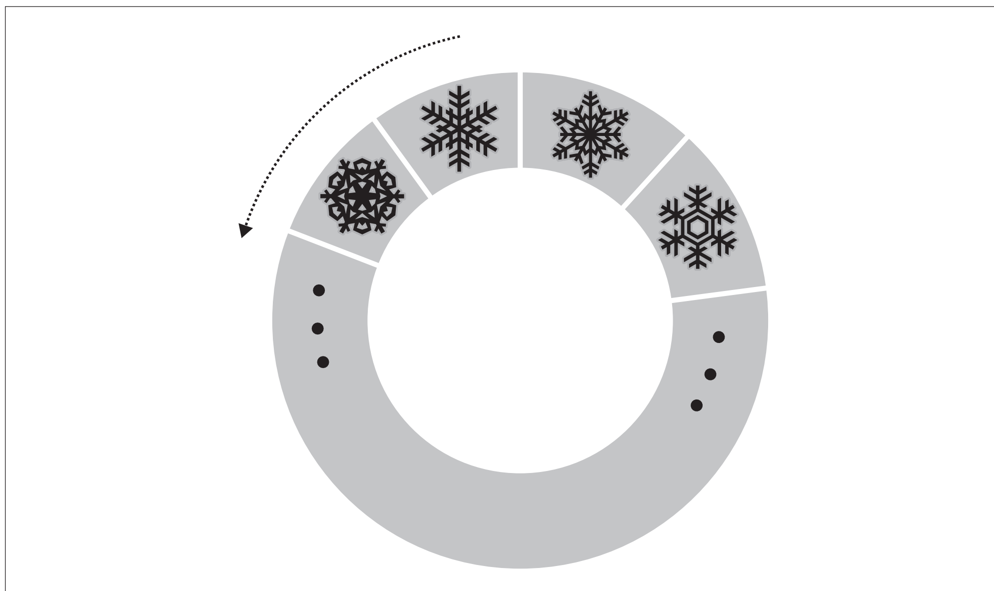


图 7-4：按照 {"$natural":-1} 排序

7.2.4　尾部游标

尾部游标是一种特殊的持久游标，这类游标不会在没有结果后销毁。游标受到 `tail -f` 命令的启发，类似地会尽可能持续地获取结果输出。因为这类游标在没有结果后也不销毁，所以一旦有新文档添加到集合里面就会被取回并输出。尾部游标只能用在固定集合上。

可惜 Mongo shell 并不支持尾部游标，但是可以看看 PHP 中的例子：

```
$cursor = $collection->find()->tailable();

while (true) {
    if (!$cursor->hasNext()) {
        if ($cursor->dead()) {
            break;
        }
        sleep(1);
    }
    else {
        while (cursor->hasNext()) {
            do_stuff(cursor->getNext());
        }
    }
}
```

游标没有销毁，要么处理结果，要么等着有更多的结果。

7.3 GridFS：存储文件

GridFS 是一种在 MongoDB 中存储大二进制文件的机制。使用 GridFS 存文件有如下几个原因。

- 利用 GridFS 可以简化需求。要是已经用了 MongoDB，GridFS 就可以不需要使用独立文件存储架构。
- GridFS 会直接利用业已建立的复制或分片机制，所以对于文件存储来说故障恢复和扩展都很容易。
- GridFS 可以避免用于存储用户上传内容的文件系统出现的某些问题。例如，GridFS 在同一个目录下放置大量的文件是没有任何问题的。
- GridFS 不产生磁盘碎片，因为 MongoDB 分配数据文件空间时以 2 GB 为一块。

7.3.1 开始使用GridFS：mongofiles

最简单的开始使用 GridFS 的方法就是利用 `mongofiles` 实用程序。`mongofiles` 内置在 MongoDB 发布版中，可以用来在 GridFS 中上传、下载、列示、查找或删除文件。像其他命令行工具一样，执行 `mongofiles --help` 可以查看可用选项。下面将会介绍如何用 `mongofiles` 从文件系统向 GridFS 上传文件，列出 GridFS 中的所有文件，下载刚上传的文件。

```
$ echo "Hello, world" > foo.txt
$ ./mongofiles put foo.txt
connected to: 127.0.0.1
added file: { _id: ObjectId('4c0d2a6c3052c25545139b88'),
              filename: "foo.txt", length: 13, chunkSize: 262144,
              uploadDate: new Date(1275931244818),
              md5: "a7966bf58e23583c9a5a4059383ff850" }
done!
$ ./mongofiles list
connected to: 127.0.0.1
foo.txt 13
$ rm foo.txt
$ ./mongofiles get foo.txt
connected to: 127.0.0.1
done write to: foo.txt
$ cat foo.txt
Hello, world
```

上面的例子中，使用了 `mongofiles` 的 3 个基本操作：`put`、`list` 和 `get`。`put` 将文件系统中的一个文件添加到 GridFS 中，`list` 会把所有添加到 GridFS 中的文件列出来，`get` 则是 `put` 的逆操作，它将 GridFS 中的文件写入到文件系统中。`mongofiles` 还支持另外两个操作：`search` 用来按文件名查找 GridFS 中的文件，`delete` 则从 GridFS 中删除一个文件。

7.3.2 通过MongoDB驱动程序操作GridFS

前面已经看到，使用命令行操作 GridFS 十分简便，使用 MongoDB 驱动程序也一样简便。例如，使用 MongoDB 的 Python 驱动程序 PyMongo，可以实现上面用 mongofiles 执行的一系列操作：

```
>>> from pymongo import Connection
>>> import gridfs
>>> db = Connection().test
>>> fs = gridfs.GridFS(db)
>>> file_id = fs.put("Hello, world", filename="foo.txt")
>>> fs.list()
[u'foo.txt']
>>> fs.get(file_id).read()
'Hello, world'
```

PyMongo 中操作 GridFS 的 API 与 mongofiles 的 API 十分类似，都可以很容易地执行基本的 put、get 和 list 操作。差不多所有 MongoDB 的驱动程序与 GridFS 打交道都遵循这个基本的模式，一般还会提供一些高级的功能。要想了解与 GridFS 有关的驱动程序特有的信息，就得查看你所使用的驱动程序的文档了。

7.3.3 内部原理

GridFS 是一个建立在普通 MongoDB 文档基础上的轻量级文件存储规范。MongoDB 服务器实际上对 GridFS 请求没什么特别照顾，所有相关工作都由客户端驱动或者工具来完成。

GridFS 的一个基本思想就是可以将大文件分成很多块，每块作为一个单独的文档存储，这样就能存大文件了。由于 MongoDB 支持在文档中存储二进制数据，可以最大限度减小块的存储开销。另外，除了存储文件本身的块，还有一个单独的文档用来存储分块的信息和文件的元数据。

GridFS 的块有个单独的集合。默认情况下，块将使用 fs.chunks 集合，如有需要可以覆盖。这个块集合里面文档的结构是非常简单的：

```
{
    "_id" : ObjectId("..."),
    "n" : 0,
    "data" : BinData("..."),
    "files_id" : ObjectId("...")
}
```

和别的 MongoDB 文档一样，块也有自己唯一的 "_id"。另外，还有些别的键。"files_id" 是包含这个块元数据的文件文档的 "_id"。"n" 表示块编号，也就是这

个块在原文件中的顺序编号。最后，"data" 包含组成文件块的二进制数据。

文件的元数据放在另一个集合中，默认是 fs.files。这里面的每个文档代表 GridFS 中的一个文件，与文件相关的自定义元数据也可以存在其中。除了用户自定义的键，GridFS 规范还定义了一些键。

- `_id`
  文件唯一的 id，在块中作为 "files_id" 键的值存储。

- `length`
  文件内容总的字节数。

- `chunkSize`
  每块的大小，以字节为单位。默认是 256 K，必要时可以调整。

- `uploadDate`
  文件存入 GridFS 的时间戳。

- `md5`
  文件内容的 md5 校验和，由服务器端生成。

在所有必需的键中，最有趣的（或者说最不太好理解的）就是这个 "md5" 了。"md5" 的值是由服务器端用 filemd5 命令生成的，用于计算上传块的 md5 校验和。也就意味着用户可以检验 "md5" 键的这个值，确保文件正确上传了。

理解了 GridFS 规范以后，实现一些驱动程序没有提供的功能就很容易了。例如，可使用 distinct 命令获取 GridFS 中不重复的文件名列表。

```
> db.fs.files.distinct("filename")
[ "foo.txt" ]
```

7.4 服务器端脚本

在服务器端可以通过 db.eval 函数来执行 JavaScript 脚本。也可以把 JavaScript 脚本保存在数据库中，然后在别的数据库命令中调用。

7.4.1 db.eval

利用 db.eval 可以在 MongoDB 的服务器端执行任意的 JavaScript 脚本。这个函数先将给定的 JavaScript 字符串传送给 MongoDB（在这里执行），然后返回结果。

db.eval 可以用来模拟多文档事务：db.eval 锁住数据库，然后执行 JavaScript，再

解锁。虽然没有内置的回滚机制，但这的确能保证一系列操作按照指定顺序发生（除非出错）。

发送代码有两种选择，或者封装进一个函数，或者不封装。下面两行代码是等价的：

```
> db.eval("return 1;")
1
> db.eval("function() { return 1; }")
1
```

只有传递参数的时候，才必须要封装成一个函数。参数通过 `db.eval` 的第二个参数传递，要写成一个数组的形式。例如，如果想给一个函数传递 `username`，可以这么做：

```
> db.eval("function(u) { print('Hello, '+u+'!'); }", [username])
```

有必要的话可以传递多个参数。例如，要计算 3 个数的和，可以这样：

```
> db.eval("function(x,y,z) { return x + y + z; }", [num1, num2, num3])
```

`num1` 对应 `x`，`num2` 对应 `y`，`num3` 对应 `z`。如果想使用可变数量的参数，调用函数时 JavaScript 会把参数存成一个数组。

`db.eval` 的表达式要是复杂的话，调试起来就需要些技巧了。JavaScript 脚本在数据库上执行，通常在出错信息里没有能帮助调试的行号。调试的一个好方法就是将调试信息写进数据库日志中，这个可以通过 `print` 函数来完成：

```
> db.eval("print('Hello, world');");
```

7.4.2　存储JavaScript

每个 MongoDB 的数据库中都有个特殊的集合，叫做 system.js，用来存放 JavaScript 变量。这些变量可以在任何 MongoDB 的 JavaScript 上下文中调用，包括 `"$where"` 子句，`db.eval` 调用，MapReduce 作业。用 `insert` 就可以将变量加进 system.js 中。

```
> db.system.js.insert({"_id" : "x", "value" : 1})
> db.system.js.insert({"_id" : "y", "value" : 2})
> db.system.js.insert({"_id" : "z", "value" : 3})
```

上例在全局作用域中定义了 `x`、`y`、`z`。现在要是想对其求和，可以这样：

```
> db.eval("return x+y+z;")
6
```

除了一些简单的值，system.js 也可以用来存放 JavaScript 代码。这样就可以很方便地自定义一些实用程序。例如，要用 JavaScript 写一个日志函数，就可以将其存放

到 system.js 中：

```
> db.system.js.insert({"_id" : "log", "value" :
... function(msg, level) {
...     var levels = ["DEBUG", "WARN", "ERROR", "FATAL"];
...     level = level ? level : 0; // check if level is defined
...     var now = new Date();
...     print(now + " " + levels[level] + msg);
... }})
```

现在，可以在任意的 JavaScript 程序中调用这个函数：

```
> db.eval("x = 1; log('x is '+x); x = 2; log('x is greater than 1', 1);");
```

数据库日志会含有类似下面这样的内容：

```
Fri Jun 11 2010 11:12:39 GMT-0400 (EST) DEBUG x is 1
Fri Jun 11 2010 11:12:40 GMT-0400 (EST) WARN x is greater than 1
```

使用存储的 JavaScript 缺点就是代码会与常规的源代码控制脱离，会搅乱客户端发送来的 JavaScript。

最适合使用存储的 JavaScript 的情况就是程序中有多个地方（也可能是不同的程序，或者不同语言的代码）都要用到一个 JavaScript 函数。将这样的函数放置在中心位置，要是有更新的话就可以不必每处都修改。要是 JavaScript 代码很长又要频繁使用的话，也可以使用存储的 JavaScript，这样存一次会节省不少网络传输时间。

7.4.3 安全性

执行 JavaScript 代码，就必须要谨慎考虑 MongoDB 的安全性。使用不慎，就会发生类似于关系型数据库的注入式攻击。好在我们能比较容易地避免这些，安全地使用 JavaScript。

若是想打印“Hello, 用户名！”给用户。其中的用户名保存在一个名为 `username` 的变量中。可以像下面这样写这段程序：

```
> func = "function() { print('Hello, "+username+"!'); }"
```

如果 `username` 是用户定义的，就可能会是这样的字符串 `"'); db.dropDatabase(); print('"`，这样代码就成了下面这样：

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!');
}"
```

整个数据库都被清干净了！

为了避免这种情况，要限定作用域。例如，在 PHP 中应该像这样写：

```
$func = new MongoCode("function() { print('Hello, "+username+"!'); }",
... array("username" => $username));
```

数据库就会安全地输出如下字符：

```
Hello, '); db.dropDatabase(); print('!
```

绝大多数驱动程序都为传递给数据库的代码提供一种特殊类型，这是因为代码实际上可以看成是一个字符串和一个作用域的组合。作用域无非就是一个保存着变量名和值映射关系的文档。当 JavaScript 函数执行的时候，这种映射就构成了函数的局部作用域。



shell 没有含有作用域的代码类型，只能对其使用字符串或者 JavaScript 函数。

7.5 数据库引用

可能 MongoDB 最鲜为人知的功能就是*数据库引用*了，也叫做 DBRef。DBRef 就像 URL，唯一确定一个到文档的引用。它自动加载文档的方式正如网站中 URL 通过链接自动加载 Web 页面一样。

7.5.1 什么是DBRef

DBRef 是个内嵌文档，就像 MongoDB 中的其他内嵌文档一样。但是 DBRef 有些必选键。下面是个简单的例子：

```
{"$ref" : collection, "$id" : id_value}
```

DBRef 指向一个集合，还有一个 `id_value` 用来在集合里面根据 `"_id"` 确定唯一的文档。这两条信息使得 DBRef 能唯一标识 MongoDB 数据库内的任何一个文档。若是想引用另一个数据库中的文档，DBRef 中有个可选键 `"$db"`，用这个就可以了：

```
{"$ref" : collection, "$id" : id_value, "$db" : database}
```



DBRef 中的键的顺序不能改变。第一个必须是 `"$ref"`，接着是 `"$id"`，然后是（可选的）`"$db"`。

7.5.2 示例模式

来看一个使用 DBRef 跨集合引用文档的例子。本例中含有两个集合，`users` 和 `notes`。

用户（user）可以创建笔记（note），笔记可以引用用户或者别的笔记。现在有一些用户文档，每一个都有唯一的用户名作为其 "_id"，以及一个独立形式的 "display_name"：

```
{"_id" : "mike", "display_name" : "Mike D"}
{"_id" : "kristina", "display_name" : "Kristina C"}
```

notes 集合稍微复杂一些。每个笔记都含有一个唯一的 "_id"。正常情况下，这个 "_id" 很可能是个 ObjectID，但是这里用整数，是为了让例子简明，突出重点。notes 还有一个 "author"，若干 "text"，以及一个可选的 "references" 指向其他笔记或者用户：

```
{"_id" : 5, "author" : "mike", "text" : "MongoDB is fun!"}
{"_id" : 20, "author" : "kristina", "text" : "... and DBRefs are easy, too",
"references": [{"$ref" : "users", "$id" : "mike"}, {"$ref" : "notes",
                                                      "$id" : 5}]}
```

第二个笔记包含一些对其他文档的引用，每一条都作为一个 DBRef 存储。应用层的程序会利用这些 DBRef 得到用户“Mike”和笔记“MongoDB is fun!”这两个文档，而它们都是与 Kristina 的笔记关联的。去引用是很容易实现的。"$ref" 的值就是要查询的集合，然后使用 "$id" 键的值，获得 "_id" 的值：

```
> var note = db.notes.findOne({"_id" : 20});
> note.references.forEach(function(ref) {
... printjson(db[ref.$ref].findOne({"_id" : ref.$id}));
... });
{ "_id" : "mike", "display_name" : "Mike D" }
{ "_id" : 5, "author" : "mike", "text" : "MongoDB is fun!" }
```

7.5.3 驱动对DBRef的支持

令人费解的是不是所有驱动程序都将 DBRef 作为普通的内嵌文档。一些驱动程序为 DBRef 提供了特殊的类型，这样就会和普通文档自动相互转换。这主要是为了开发者提供便利，因为这样可以忽略少量细节。例如，使用 PyMongo 中的 DBRef 类型可以像下面这样表示上面的例子：

```
>>> note = {"_id": 20, "author": "kristina",
...         "text": "... and DBRefs are easy, too",
...         "references": [DBRef("users", "mike"), DBRef("notes", 5)]}
```

当保存时，DBRef 实例会自动被转换成等价的内嵌文档。当作为查询结果返回时，逆操作也会自动进行，就又得到了 DBRef 的实例。

一些驱动程序还添加了别的辅助工具来操作 DBRef，比如处理去引用的方法，甚至提供当返回结果包含引用时自动去引用的机制。这些辅助功能随驱动程序的不同而变化，要想知道最新的信息，需要参考具体驱动程序的文档。

7.5.4 什么时候该使用DBRef呢

在 MongoDB 中表示这种对其他文档的引用关系，并不是非 DBRef 不可。事实上，即便前面的例子也使用了些不同的机制做引用：每个笔记的 `"author"` 键仅存储了 author 文档的 `"_id"` 键。没有必要使用 DBRef，因为已经知道每个 author 就是 users 集合里面的一个文档。这种类型的引用以前也出现过：在 GridFS 的块文档中 `"files_id"` 键仅仅就是对文件文档 `"_id"` 的引用。知道这一点，每次要保存引用的时候就得做抉择了：用 DBRef 呢，还是只存储 `"_id"` 呢？

保存 `"_id"` 相当不错，因为会更加紧凑，对开发者而言也更轻量。但另一方面，DBRef 能够引用任意集合（甚至任意数据库）的文档，开发者不必知道和记住被引用的文档在哪些集合里面。驱动程序和一些工具对 DBRef 提供些额外的功能（比如自动去引用），而且服务器端在日后也可能会有更高级的支持。

总之，存储一些对不同集合的文档的引用时，最好使用 DBRef，就像前面的例子。或者想使用驱动程序或者工具中 DBRef 特有的功能，只能用 DBRef 了。否则，最好存储 `"_id"` 作为引用来使用，因为这样更精简，也更容易操作。

第 8 章 管理

管理 MongoDB 还算是轻松的。无论简单的备份还是做带有复制的多节点系统，都有快捷的方法。这也体现了 MongoDB 的设计理念：尽可能简化系统操作。系统会尽量自动完成各种配置，而不是让用户和管理员做这做那。即便如此，还是有少量管理任务需要手工干预。

本章从开发者的视角切换到管理员的视角，看看 MongoDB 如何运维。你可能在一个小型团队中负责开发和运维，或者你是一个 DBA 想研究如何使用 MongoDB，那么本章恰好适合你阅读。

本章主要内容如下。

- MongoDB 就是一个普通的命令行程序，用 mongod 调用。
- MongoDB 提供了内置的管理接口和监控功能，易于与第三方监控包集成。
- MongoDB 支持基本的、数据库级别的用户认证，包括只读用户，以及独立的管理员权限。
- 有多种方式备份 MongoDB，主要取决于实际的情况该用哪种。

8.1 启动和停止MongoDB

在第 2 章中，已经介绍了启动 MongoDB 的基本方式。这里会更加细致地介绍管理员在生产环境中部署 Mongo 的要点。

8.1.1 从命令行启动

执行 mongod，启动 MongoDB 服务器。mongod 有很多可配置的启动选项：在命令

行运行 `mongod --help` 可以查看所有选项。一些主要选项如下。

- `--dbpath`

  指定数据目录；默认值是 /data/db/（Windows 下是 C:\data\db\）。每个 `mongod` 进程都需要独立的数据目录，所以要是有 3 个 `mongod` 实例，必须要有 3 个独立的数据目录。当 `mongod` 启动时，会在数据目录中创建 mongod.lock 文件，这个文件用于防止其他 `mongod` 进程使用该数据目录。如果使用同一个数据目录启动另一个 MongoDB 服务器，则会报错：

  `"Unable to acquire lock for lockfilepath: /data/db/mongod.lock."`

- `--port`

  指定服务器监听的端口号。默认端口是 27017，是个其他进程不怎么用的端口（除了其他 `mongod` 进程）。要是运行多个 `mongod` 进程，则要给每个指定不同的端口号。如果启动 `mongod` 时端口被占用，则报错：

  `"Address already in use for socket: 0.0.0.0:27017"`

- `--fork`

  以守护进程的方式运行 MongoDB，创建服务器进程。

- `--logpath`

  指定日志输出路径，而不是输出到命令行。如果对文件夹有写权限的话，系统会在文件不存在时创建它。它会将已有文件覆盖掉，清除所有原来的日志记录。如果想保留原来的日志，还需要使用 `--logappend` 选项。

- `--config`

  指定配置文件，加载命令行未指定的各种选项。详见 8.1.2 节。

现在启动 MongoDB 服务器，让其作为守护进程监听 5586 号端口，并将所有输出记录到 mongodb.log：

```
$ ./mongod --port 5586 --fork --logpath mongodb.log
forked process: 45082
all output going to: mongodb.log
```

当初次安装并启动 MongoDB 时，最好看看日志。这是人们经常忽视的一点，尤其是当 MongoDB 使用开机启动脚本启动的时候。但是日志经常会有些重要的警告信息，能够帮助避免发生一些错误。要是启动 MongoDB 时没有任何警告，则万事大吉。不过实际情况下你可能看到下面这些信息：

```
$ ./mongod
```

```
Sat Apr 24 11:53:49 Mongo DB : starting : pid = 18417 port = 27017
dbpath = /data/db/ master = 0 slave = 0  32-bit
****
WARNING: This is development version of MongoDB.
         Not recommended for production.
****

** NOTE: when using MongoDB 32 bit, you are limited to about
**       2 gigabytes of data see
**       http://blog.mongodb.org/post/137788967/32-bit-limitations
**       for more

Sat Apr 24 11:53:49 db version v1.5.1-pre-, pdfile version 4.5
Sat Apr 24 11:53:49 git version: f86d93fd949777d5fbe00bf9784ec0947d6e75
b9
Sat Apr 24 11:53:49 sys info: Linux ubuntu 2.6.31-15-generic ...
Sat Apr 24 11:53:49 waiting for connections on port 27017
Sat Apr 24 11:53:49 web admin interface listening on port 28017
```

这里运行的 MongoDB 是个开发版，要是用稳定版就不会有第一个警告了。第二个警告是因为用的是 32 位的 MongoDB。在 32 位下，MongoDB 只能处理 2 GB 的数据，这是因为 MongoDB 使用内存映射文件存储引擎（附录 C 介绍了 MongoDB 存储引擎的相关内容）。要是在 64 位机器上使用稳定版，就不会有这些警告了，但最好要弄明白 MongoDB 日志的原理并养成看日志的习惯。

日志的这部分开头即便重启也没什么变化，所以如果明白了其中的含义以后，在开机脚本中启动 MongoDB 可以放心地忽略这部分。然而，最好要在每次安装、升级，宕机恢复后再次确认一下 MongoDB 和系统都运转良好。

8.1.2 配置文件

MongoDB 支持从文件获取配置信息。当需要的配置非常多或者要自动化 MongoDB 的启动时就会用到这个。指定配置文件可以用 `-f` 或者 `--config` 选项。例如，运行 `mongod --config ~/.mongodb.conf` 就会使用 `~/.mongodb.conf` 作为配置文件。

配置文件和命令行的功能是完全一样的。下面就是一个配置文件的例子：

```
# Start MongoDB as a daemon on port 5586

port = 5586
fork = true # daemonize it!

logpath = mongodb.log
```

这个配置文件指定的选项和我们之前使用常规的命令行参数时所使用的相同。它还体现了配置文件的一些特点。

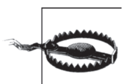
- 以 # 开头的行是注释。
- 指定选项的语法就是这种“选项 = 值”的形式，其中选项是区分大小写的。
- 命令行中那些如 `--fork` 的开关选项，其值要设为 `true`。

8.1.3 停止MongoDB

让 MongoDB 稳妥地停下来和启动它同样重要。有很多途径可以有效地做到这点。

最基本的方法就是向 MongoDB 服务器发送一个 SIGINT 或者 SIGTERM 信号。如果服务器是作为前台进程运行在终端的，就直接按 Ctrl-C。否则，就用 `kill` 这种命令发出信号。如果 `mongod` 的 PID 是 10014，就可以 `kill -2 10014`（SIGINT）或者 `kill 10014`（SIGTERM）。

当 `mongod` 收到 SIGINT 或者 SIGTERM 时，会稳妥退出。也就是说会等到当前运行的操作或者文件预分配完成（需要一些时间），关闭所有打开的连接，将缓存的数据刷新到磁盘，最后停止。



千万不要向运行中的 MongoDB 发送 SIGKILL（`kill-9`）。这样会导致数据库直接关闭，上面讲到的步骤都将被忽略，这会使数据文件损毁。要是真的发生了不幸，一定要在启动备份之前修复数据库[1]（详见 8.4.5 节）。

另一种稳妥的方式就是使用 `shutdown` 命令，`{"shutdown" : 1}`。这是管理命令，要在 admin 数据库下使用。shell 提供了辅助函数，来简化这一过程：

```
> use admin
switched to db admin
> db.shutdownServer();
server should be down...
```

8.2 监控

作为 MongoDB 管理员，很重要的工作就是监控系统的状态和性能。好在 MongoDB 有很多功能，使得监控很容易。

8.2.1 使用管理接口

默认情况下，启动 `mongod` 时还会启动一个（非常）基本的 HTTP 服务器，该服务器监听的端口号比主服务的端口号大 1000。这个服务器提供了 HTTP 接口，可以查看 MongoDB 的一些基本信息。这些呈现的信息也能通过 shell 来查看，不过 HTTP 接口提供的信息更加易读。

译注1：否则下次将无法正常启动。

启动服务器，然后在浏览器里查看 http://localhost:28017，就能看见管理接口。（如果用 `--port` 指定了端口，则要用比它大 1000 的端口号）。你会看到如图 8-1 所示的页面。

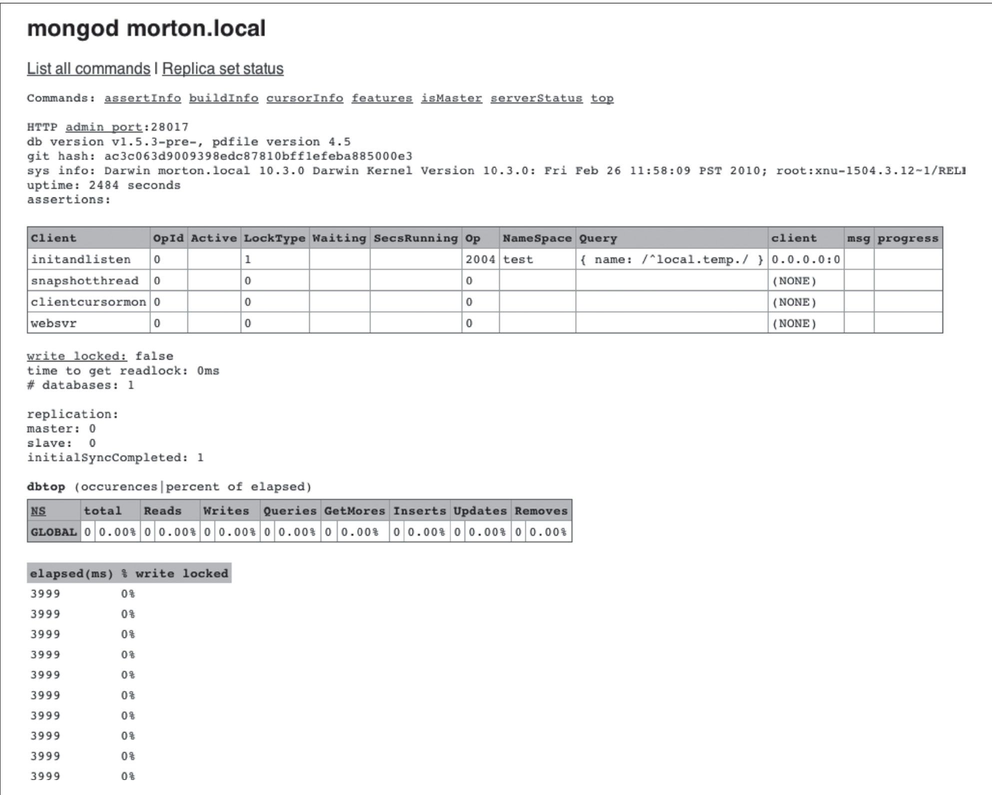
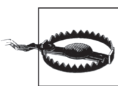


图 8-1：管理接口

可以看到断言、锁、索引和复制等相关信息。还有些更加常见的信息，如日志前导、数据库命令列表。

要想利用好管理接口（比如，访问命令列表），需要用 `--rest` 选项开启 REST 支持。也可以在启动 `mongod` 时使用 `--nohttpinterface` 关闭管理接口。



不要用驱动程序连接 HTTP 接口，也不要通过 HTTP 连接本机驱动端口。驱动端口只能处理本机 MongoDB 传输协议，不能处理 HTTP 请求。例如，如果在浏览器中查看 http://localhost:27017，会看到如下提示：

```
You are trying to access MongoDB on the native driver port.
 For http diagnostic access, add 1000 to the port number
```

同样，也不能用本机 MongoDB 传输协议去访问管理接口的端口。

8.2.2 serverStatus

要获取运行中的 MongoDB 服务器统计信息，最基本工具就是 serverStatus 命令，输出如下所示（不同平台不同版本的键会有差异）：

```
> db.runCommand({"serverStatus" : 1})
{
    "version" : "1.5.3",
    "uptime" : 166,
    "localTime" : "Thu Jun 10 2010 15:47:40 GMT-0400 (EDT)",
    "globalLock" : {
        "totalTime" : 165984675,
        "lockTime" : 91471425,
        "ratio" : 0.551083556358441
    },
    "mem" : {
        "bits" : 64,
        "resident" : 101,
        "virtual" : 2824,
        "supported" : true,
        "mapped" : 336
    },
    "connections" : {
        "current" : 141,
        "available" : 19859
    },
    "extra_info" : {
        "note" : "fields vary by platform"
    },
    "indexCounters" : {
        "btree" : {
            "accesses" : 1563,
            "hits" : 1563,
            "misses" : 0,
            "resets" : 0,
            "missRatio" : 0
        }
    },
    "backgroundFlushing" : {
        "flushes" : 2,
        "total_ms" : 44,
        "average_ms" : 22,
        "last_ms" : 36,
        "last_finished" : "Thu Jun 10 2010 15:46:54 GMT-0400 (EDT)"
    },
    "opcounters" : {
        "insert" : 38195,
        "query" : 8874,
        "update" : 4058,
        "delete" : 389,
        "getmore" : 888,
        "command" : 17731
    },
    "asserts" : {
```

```
        "regular" : 0,
        "warning" : 0,
        "msg" : 0,
        "user" : 5054,
        "rollovers" : 0
    },
    "ok" : true
}
```



原始的统计信息同样可以由 HTTP 接口以 JSON 的形式得到，位置在 /_status（http://localhost:28017/_status）。不但包括 serverStatus 输出，还有其他一些有用命令的输出。详见 8.2.1 节。

serverStatus 呈现了 MongoDB 内部的详细信息。比如当前服务器版本、运行时间（以秒计）、当前连接数。有些信息还需要解释一下。

"globalLock" 的值表示全局写入锁占用了服务器多少时间（以微秒计）。"mem" 包含服务器内存映射了多少数据，服务器进程的虚拟内存和常驻内存的占用情况（单位是 MB）。"indexCounters" 表示 B 树在磁盘检索（"misses"）和内存检索（"hits"）的次数。如果这个比值开始上升就要考虑添加内存了，否则系统性能就会受到影响。"backgroudFlushing" 表示后台做了多少次 fsync 以及用了多少时间。"opcounters" 文档非常重要，包含了每种主要操作的次数。最后，"asserts" 统计了断言的次数。

serverStatus 结果中的所有计数都是在服务器启动时开始计算的，如果过大就会复位。当发生复位时，所有计数器都复位，"asserts" 中的 "rollovers" 值会增加。

8.2.3 mongostat

serverStatus 虽然强大，但对监控服务器来说却不怎么易用。还好，MongoDB 还提供了 mongostat，可以便捷地查看 serverStatus 的结果。

mongostat 输出一些 serverStatus 提供的重要信息。它会每秒钟输出新的一行，比之前看到的静态计数实时性更好。它输出多个列，分别是 inserts/s、commands/s、vsize 和 % locked，与 serverStatus 的数据相对应。

8.2.4 第三方插件

绝大多数管理员可能已经使用监控系统跟踪服务器的运行情况。serverStatus 和 /_status URL 的出现使得编写 MongoDB 的监控插件非常容易。写作本书时，好多监控系统都有了 MongoDB 插件，例如 Nagios、Munin、Ganglia、Cacti。登录 http://dochub.mongodb.org/core/monitoring 查看与监控工具有关的文档。

8.3 安全和认证

系统管理员的一项重要工作就是确保系统的安全。使 MongoDB 安全的最好方法就是在一个可信的环境中运行它，保证只有可信的机器才能访问它。MongoDB 支持对单个连接的认证，即便这个认证的权限模式很简陋。

8.3.1 认证的基础知识

每个 MongoDB 实例中的数据库都可以有许多用户。如果开启了安全性检查，则只有数据库认证用户才能执行读或者写操作。在认证的上下文中，MongoDB 会将普通的数据作为 admin 数据库处理。admin 数据库中的用户被视为超级用户（即管理员）。在认证之后，管理员可以读写所有数据库，执行特定的管理命令，如 `listDatabases` 和 `shutdown`。

在开启安全检查之前，一定要至少有个管理员账号。请看下面的例子，开始时 shell 连接的是没有开启安全检查的服务器：

```
> use admin
switched to db admin
> db.addUser("root", "abcd");
{
    "user" : "root",
    "readOnly" : false,
    "pwd" : "1a0f1c3c3aa1d592f490a2addc559383"
}
> use test
switched to db test
> db.addUser("test_user", "efgh");
{
    "user" : "test_user",
    "readOnly" : false,
    "pwd" : "6076b96fc3fe6002c810268702646eec"
}
> db.addUser("read_only", "ijkl", true);
{
    "user" : "read_only",
    "readOnly" : true,
    "pwd" : "f497e180c9dc0655292fee5893c162f1"
}
```

上面添加了管理员 root，在 test 数据库添加了两个普通帐号。其中一个有只读权限，不能对数据库写入。在 shell 中创建只读用户只要将 `addUser` 的第 3 个参数设为 `true` 就可以了。调用 `addUser` 必须有相应数据库的写权限。这里可以对所有数据库调用 `addUser`，因为还没有开启安全检查。



addUser 不仅能添加用户，它还能修改用户口令或者只读状态。所以设置用户名、新密码或者只读属性都交给 addUser 好了。

现在重启服务器，这次加入 --auth 命令行选项，开启安全检查。之后，通过 shell 重新连接数据库：

```
> use test
switched to db test
> db.test.find();
error: { "$err" : "unauthorized for db [test] lock type: -1 " }
> db.auth("read_only", "ijkl");
1
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
> db.test.insert({"x" : 2});
unauthorized
> db.auth("test_user", "efgh");
1
> db.test.insert({"x": 2});
> db.test.find();
{ "_id" : ObjectId("4bb007f53e8424663ea6848a"), "x" : 1 }
{ "_id" : ObjectId("4bb0088cbe17157d7b9cac07"), "x" : 2 }
> show dbs
assert: assert failed : listDatabases failed:{
    "assertion" : "unauthorized for db [admin] lock type: 1
",
    "errmsg" : "db assertion failure",
    "ok" : 0
}
> use admin
switched to db admin
> db.auth("root", "abcd");
1
> show dbs
admin
local
test
```

第一次连接时，不能对 test 数据库执行任何操作（无论读写）。作为 read_only 用户认证之后，就能查找了。但插入数据时，由于权限不足还是遇到了问题。test_user 不是只读的，所以能正常插入数据。但作为一个非特权用户，test_user 不能使用 show dbs 帮助程序来列举所有数据库。最后作为管理员 root 认证后，就能对所有数据库执行任意操作了。

8.3.2 认证的工作原理

数据库的用户账号以文档的形式存储在 system.users 集合里面。文档的结构

是{"user" : username, "readOnly": true, "pwd" : password hash}。password hash 是根据用户名和密码生成的散列。

知道了用户信息是如何存储的以及存储位置后，有些日常管理任务执行起来就很轻松了。例如，在 system.users 集合中删掉用户账号文档，就可以删除用户：

```
> db.auth("test_user", "efgh");
1
> db.system.users.remove({"user" : "test_user"});
> db.auth("test_user", "efgh");
0
```

用户认证时，服务器将认证和连接绑定来跟踪认证。也就是说如果驱动程序或是工具使用了连接池或是因故障切换到另一个节点，所有认证用户必须对每个新连接重新认证。有的驱动程序能够将这步透明化，但要是没有，就得手动完成。真是这样的话，或许应该不用 --auth（通过将 MongoDB 部署到可信环境中然后在客户端处理认证）。

8.3.3　其他安全考虑

除了认证还有许多选项值得考虑，来锁定 MongoDB 实例。首先即便用了认证，MongoDB 传输协议也是不加密的。如果需要加密，可以用 SSH 隧道或者类似的技术做客户端和服务器之间的加密。

这里建议将 MongoDB 服务器布置在防火墙后或者布置在只有应用服务器能访问的网络中。但要是 MongoDB 必须能被外面访问到的话，建议使用 --bindip 选项，可以指定 mongod 绑定到的本地 IP 地址。例如，只能从本机应用服务器访问，可以运行 "mongod --bindip localhost"。

8.2.1 节已经讲过，默认情况下 MongoDB 会开启一个简单的 HTTP 服务器，便于查看运行、锁、复制等方面的信息。要是不想公开这些信息，就应该通过 --nohttpinter-face 将管理接口关闭。

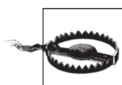
最后，还可以用 --noscripting 完全禁止服务端 JavaScript 的执行。

8.4　备份和修复

做备份是管理任何数据存储系统的一项非常重要的任务。恰当地备份往往很难，搞不好会弄巧成拙，还不如不备份呢。还好，MongoDB 提供了一些选项让这个过程不再困难重重。

8.4.1 数据文件备份

MongoDB 将所有数据都存放在数据目录下。默认目录是 /data/db/（Windows 下是 C:\data\db\）。启动 MongoDB 的时候可以用 `--dbpath` 指定数据目录。不论数据目录在哪里，它都存放着 MongoDB 的所有数据。也就是说，要想备份 MongoDB，只要简单创建数据目录中所有文件的副本就可以了。



除非服务器做了完整的 `fsync`，还不允许写入，否则在运行 MongoDB 时创建数据目录的副本并不安全。这样的备份很可能已经破损了，需要修复（详见 8.4.5 节）。

在运行 MongoDB 时复制数据目录不太安全，所以就得先把服务器关了，再复制数据目录。假设服务器安全关闭了（详见 8.1 节），数据库目录中就是关闭那一刻数据的快照。在服务器重新启动之前，可以复制目录作为备份。

虽然关停服务器再复制数据目录做备份很有效，也很安全，但还是不太理想。本章剩下的部分会介绍不需要停机的备份方式。

8.4.2 mongodump和mongorestore

`mongodump` 就是一种能在运行时备份的方法，MongoDB 自带这个工具。`mongodump` 对运行的 MongoDB 做查询，然后将所有查到的文档写入磁盘。因为 `mongodump` 是一般的客户端，所以可供运行的 MongoDB 使用，即便是正在处理其他请求或是执行写入也没有问题。



`mongodump` 使用普通的查询机制，所以产生的备份不一定是服务器数据的实时快照。服务器在备份过程中处理写入时尤为明显。

`mongodump` 还带来个问题，备份时的查询会对其他客户端的性能产生不利影响。

像大多数 MongoDB 的命令行工具一样，`mongodump` 也可以通过运行 `--help` 选项查看所有选项：

```
$ ./mongodump --help
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
                          verbosity e.g. -vvvvv)
  -h [ --host ] arg       mongo host to connect to ("left,right" for pairs)
  -d [ --db ] arg         database to use
  -c [ --collection ] arg collection to use (some commands)
  -u [ --username ] arg   username
```

```
  -p [ --password ] arg    password
  --dbpath arg             directly access mongod data files in the given
path,
                           instead of connecting to a mongod instance - needs
                           to lock the data directory, so cannot be used if a
                           mongod is currently accessing the same path
  --directoryperdb         if dbpath specified, each db is in a separate
                           directory
  -o [ --out ] arg (=dump) output directory
```

除了 mongodump，MongoDB 还提供了从备份中恢复数据的工具 mongorestore。mongorestore 获取 mongodump 的输出结果，并将备份的数据插入到运行的 MongoDB 实例中。下面的例子演示了从数据库 test 到 backup 目录的热备份，接着还调用了 mongorestore：

```
$ ./mongodump -d test -o backup
connected to: 127.0.0.1
DATABASE: test   to     backup/test
    test.x to backup/test/x.bson
         1 objects
$ ./mongorestore -d foo --drop backup/test/
connected to: 127.0.0.1
backup/test/x.bson
     going into namespace [foo.x]
     dropping
     1 objects
```

上面的例子中，-d 指定了要恢复的数据库，这里是 foo。这个选项可以将备份恢复到与原来不同名的数据库中。--drop 代表在恢复前删除集合（若存在）。否则，数据就会与现有集合数据合并，可能会覆盖一些文档。再强调一次，要获得完整的选项列表，可以运行 mongorestore --help。

8.4.3 fsync和锁

虽然用 mongodump 和 mongorestore 能不停机备份，但是我们却失去了获取实时数据视图的能力。MongoDB 的 fsync 命令能在 MongoDB 运行时复制数据目录还不会损毁数据。

fsync 命令会强制服务器将所有缓冲区写入磁盘。还可以选择上锁阻止对数据库的进一步写入，直到释放锁为止。写入锁是让 fsync 在备份时发挥作用的关键。下面的例子展示了如何在 shell 中操作，强制执行了 fsync 并获得了写入锁：

```
> use admin
switched to db admin
> db.runCommand({"fsync" : 1, "lock" : 1});
{
    "info":"now locked against writes, use db.$cmd.sys.unlock.findOne() to
unlock",
    "ok" : 1
}
```

至此，数据目录的数据就是一致的，且为数据的实时快照。因为上了写入锁，可以安全地将数据目录副本用做备份。要是数据库运行在有快照功能的文件系统上时，比如 LVM[1] 或者 EBS[2]，这个就很有用了，因为拍个数据库目录快照非常之快。

备份好了，就要解锁：

```
> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
> db.currentOp();
{ "inprog" : [ ] }
```

运行 currentOp 是为了确保已经解锁了。（初次请求解锁会花点时间。）

有了 fsync 命令，就能非常灵活地备份，不用停掉服务器，也不用牺牲备份的实时特性。要付出的代价就是一些写入操作被暂时阻塞了。唯一不耽误读写还能保证实时快照的备份方式就是通过从服务器备份。

8.4.4 从属备份

虽然上面说的几种方式在备份数据方面已经很灵活了，但是都不及在从服务器上备份。当以复制的方式运行 MongoDB 时（详见第 9 章），前面提到的备份技术就不仅能用在主服务器上，也可以用在从服务器上，而且效果还会更好。从服务器的数据几乎与主服务器同步。因为不太在乎从属服务器的性能或是能不能读写，于是就能随意选择上面的 3 种备份方式：关停、转储和恢复工具或 fsync 命令。在从服务器上备份是 MongoDB 推荐的备份方式。

8.4.5 修复

做备份是为了以备不测，比如停电，甚至大象闯入数据中心什么的，不管怎样数据都是安全的。总是有机器宕掉，又恰巧没有备份（或者没有可以转移故障的从服务器）这种倒霉时刻。要是停电或者软件崩溃，恢复后机器的磁盘一般没有问题。但 MongoDB 的存储方式不能保证磁盘上的数据还能用，因为可能有损毁（关于 MongoDB 的存储引擎详见附录 C）。幸好，MongoDB 内置的修复功能会试着恢复损坏的数据文件。

未能正常停止 MongoDB 后应该修复数据库。要是未正常停止，下次启动服务器备份时 MongoDB 会提示：

```
**************
old lock file: /data/db/mongod.lock. probably means unclean shutdown
recommend removing file and running --repair
see: http://dochub.mongodb.org/core/repair for more information
*************
```

注1：Linux的逻辑卷管理器。
注2：Elastic Block Store，亚马逊提供的一种持久存储方案。

修复所有数据库最简单的方式就是加上 --repair：mongod --repair 来启动服务器。修复数据库的实际过程实际上非常简单：将所有的文档导出然后马上导入，忽略那些无效的文档。完成以后，会重新建立索引。了解这一机制对理解修复的一些属性有帮助。数据量大的话会花很多时间，因为所有数据都要验证，所有索引都要重建。修复后可能会比修复前少些文档，因为损毁的文档都被丢弃了[1]。



修复数据库还能起到压缩数据的作用。闲置的空间（比如删除体积较大的集合，或删除大量文档后腾出的空间）在修复后被重新回收。

修复运行中的服务器上的数据库，要在 shell 中用 repairDatabase。使用下列方法修复一下 test 数据库：

```
> use test
switched to db test
> db.repairDatabase()
{ "ok" : 1 }
```

要是不通过 shell 而是通过驱动程序，可以用 repairDatabase 来完成相同的事情：

```
{"repairDatabase" : 1}
```

修复损毁的数据是不得已时的最后一招。尽可能稳妥地停掉服务器，利用复制功能实现故障恢复，经常做备份，这些才是最有效的管理数据的手段。

译注1：MongoDB 1.8以后引入了日志系统，使得系统恢复时间大大缩短。

第 9 章 复制

MongoDB 管理员最重要的工作莫过于确保复制设置正确，且运转良好。这里强烈推荐在生产环境中使用 MongoDB 的复制功能，尤其是现在的存储引擎还不支持单机持久性（详见目录 C）。不仅可以用复制来应对故障切换、数据集成，还可以用来做读扩展、热备份或作为离线批处理的数据源。本章会介绍复制的方方面面。

9.1 主从复制

主从复制是 MongoDB 最常用的复制方式。这种方式非常灵活，可用于备份、故障恢复、读扩展等（详见图 9-1 和图 9-2）。

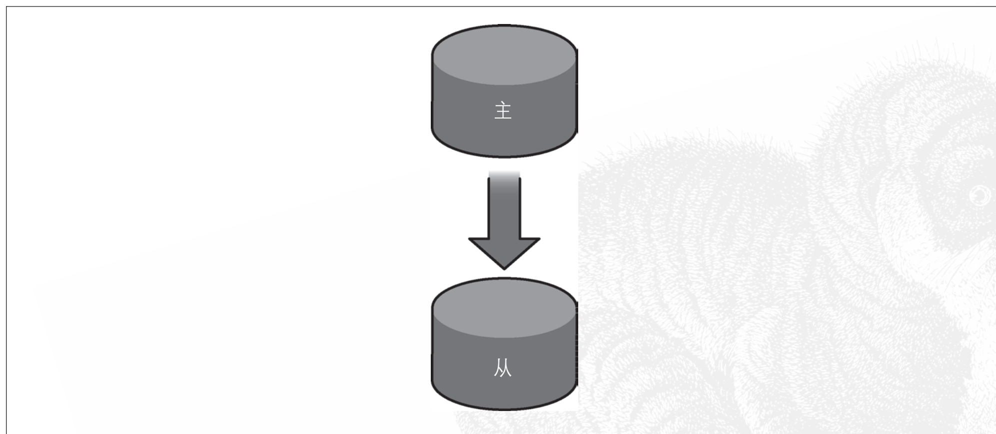


图 9-1：搭配一个从节点的主节点

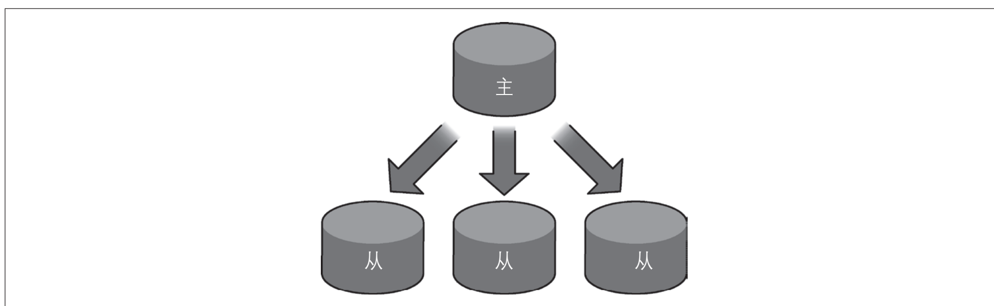


图 9-2：搭配 3 个从节点的主节点

最基本的设置方式就是建立一个主节点和一个或者多个从节点，每个从节点要知道主节点的地址。运行 mongod --master 就启动了主服务器。运行 mongod --slave --source *master_address* 则启动了从服务器，其中 *master_address* 就是上面主节点的地址。

生产环境下会有多台服务器的，不过这里简化一下就在同一台机器上试验了。首先，给主节点建立数据目录，并绑定端口（10000）：

```
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000 --master
```

接着设置从节点，记着要选择不同的目录和端口，并且用 --source 为从节点指明主节点的地址：

```
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001 --slave --source
localhost:10000
```

所有从节点都从主节点复制内容。目前还没有能够从从节点复制的机制（菊花链），原因就是从节点并不保存自己的 oplog（关于 oplog，详见 9.4 节）。

一个集群中有多少个从节点并没有明确的限制，但是上千个从节点对单个主机点发起查询也会让其吃不消的。所以实际中，不超过 12 个从节点的集群就可以运转良好了。

9.1.1 选项

主从复制有些有用的选项。

- `--only`
  在从节点上指定只复制特定某个数据库（默认复制所有数据库）。

- `--slavedelay`
  用在从节点上，当应用主节点的操作时增加延时（单位是秒）。这样就能轻松设

置延时从节点了，这种节点对用户无意中删除重要文档或者插入垃圾数据等事故有很重要的防护作用。这些不良操作都会被复制到所有从节点上。通过延缓执行操作，可以有个恢复的时间差。

- `--fastsync`
以主节点的数据快照为基础启动从节点。如果数据目录一开始是主节点的数据快照，从节点用这个选项启动要比做完整同步快很多。

- `--autoresync`
如果从节点与主节点不同步了，则自动重新同步。（详见 9.4 节。）

- `--oplogSize`
主节点 oplog 的大小（单位是 MB）。（详见 9.4 节。）

9.1.2　添加及删除源

启动从节点时可以用 `--source` 指定主节点，也可以在 shell 中配置这个源。

假设主节点绑定了 localhost:27017。启动从节点时可以不添加源，而是随后向 `sources` 集合添加主节点信息：

```
$ ./mongod --slave --dbpath ~/dbs/slave --port 27018
```

现在可以在 shell 中运行如下命令，将 localhost:27017 作为源添加到从节点上：

```
> use local
> db.sources.insert({"host" : "localhost:27017"})
```

看看从属节点的日志，会发现它与 localhost:27017 同步。

在 `sources` 集合中插入源后，如果立刻进行查询就能查到插入的文档：

```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017"
}
```

当完成同步后，该文档就被更新了：

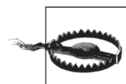
```
> db.sources.find()
{
    "_id" : ObjectId("4c1650c2d26b84cc1a31781f"),
    "host" : "localhost:27017",
    "source" : "main",
    "syncedTo" : {
        "t" : 1276530906000,
        "i" : 1
```

```
        },
        "localLogTs" : {
            "t" : 0,
            "i" : 0
        },
        "dbsNextPass" : {
            "test_db" : true
        }
}
```

假设在生产环境下，想更改从节点的配置，改用 prod.example.com 为源，则可以用 insert 和 remove 来完成：

```
> db.sources.insert({"host" : "prod.example.com:27017"})
> db.sources.remove({"host" : "localhost:27017"})
```

可以看到，sources 集合可以被当做普通集合进行操作，而且为管理从节点提供了很大的灵活性。



要是切换的两个主节点有相同的集合，MongoDB 会尝试合并，但不保证能正确合并。要是使用的一个从节点对应多个不同的主节点，最好在主节点上使用不同的命名空间。

9.2　副本集

简单地说，副本集（Replica Set）就是有自动故障恢复功能的主从集群。主从集群和副本集最为明显的区别是副本集没有固定的“主节点”：整个集群会选举出一个“主节点”，当其不能工作时则变更到其他节点。然而，二者看上去非常相似：副本集总会有一个活跃节点（primary）和一个或多个备份节点（secondary）。详见图 9-3 ～图 9-5。

副本集最美妙的地方就是所有东西都是自动化的。首先，它为你做了很多管理工作，自动提升备份节点成为活跃节点，以确保运转正常。其次，它对于开发者而言，也很易用：仅需要为副本集指定一下服务器，驱动程序就会自动找到服务器，在当前活跃节点死机时自动处理故障恢复这类事情。

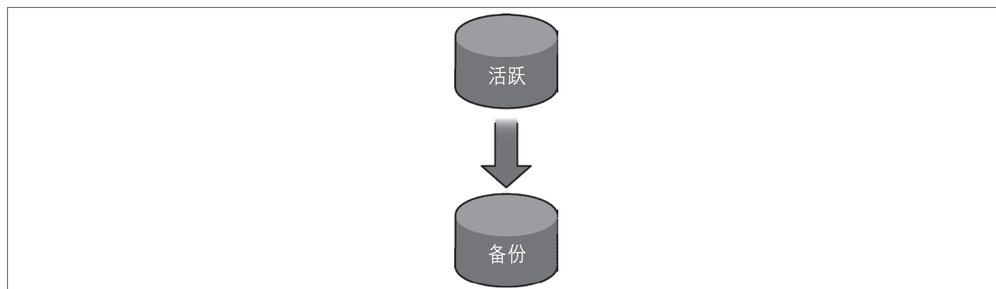


图 9-3：包含两个成员的副本集

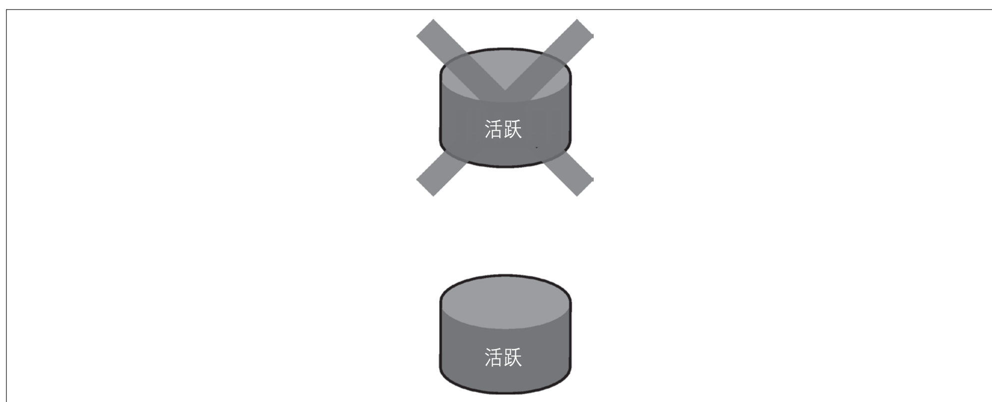


图 9-4：活跃节点不工作了，备份节点就会成为活跃节点

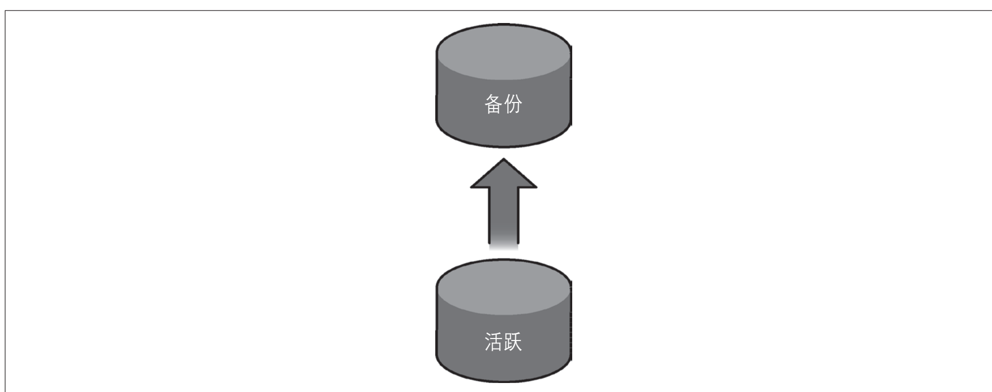


图 9-5：如果原来的活跃节点恢复了，它会成为新的活跃节点的备份节点

9.2.1 初始化副本集

设置副本集比设置主从集群稍微复杂一点。先从最简单的例子开始：两个服务器。



不能用 `localhost` 地址作为成员，所以得找到机器的主机名。在 *NIX 系统中，可以这样：

```
$ cat /etc/hostname
morton
```

首先，要为每一个服务器创建数据目录，选择端口：

```
$ mkdir -p ~/dbs/node1 ~/dbs/node2
```

在启动之前，还得做个重要决定：给副本集起个名字。名字是为了易于与别的副本集区分，也是为了方便地将整个集合视为一个整体。这里就命名为 `"blort"`。

之后就启动服务器。`--replSet` 是个没接触过的选项，作用是让服务器知晓在这个

"blort" 副本集中还有别的同伴，位置在 morton:10002（还没启动呢）：

```
$ ./mongod --dbpath ~/dbs/node1 --port 10001 --replSet blort/morton:10002
```

以同样的方式启动另一台：

```
$ ./mongod --dbpath ~/dbs/node2 --port 10002 --replSet blort/
morton:10001
```

如果想添加第 3 台，下面两种方式都行：

```
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/
morton:10001
$ ./mongod --dbpath ~/dbs/node3 --port 10003 --replSet blort/morton:10001,
                                                            morton:10002
```

副本集的一个亮点就是有自检测功能：在其中指定单台服务器后，MongoDB 就会自动搜索并连接其余的节点。

启动了几台服务器之后，日志就会告诉你副本集没有进行初始化。因为还差最后一步：在 shell 中初始化副本集。

在 shell，连接其中一个服务器（下面的例子中使用 morton:10001）。初始化命令只能执行一次：

```
$ ./mongo morton:10001/admin
MongoDB shell version: 1.5.3
connecting to localhost:10001/admin
type "help" for help
> db.runCommand({"replSetInitiate" : {
... "_id" : "blort",
... "members" : [
...     {
...         "_id" : 1,
...         "host" : "morton:10001"
...     },
...     {
...         "_id" : 2,
...         "host" : "morton:10002"
...     }
... ]}})
{
        "info" : "Config now saved locally.  Should come online in about a
                  minute.",
        "ok" : true
}
```

这个初始化文档略显复杂，但一点点来看还是可以理解的。

`"_id" : "blort"`
副本集的名字。

`"members" : [...]`

副本集中的服务器列表。过后还能添加。每个服务器文档至少有两个键。

`"_id" : N`

每个服务器的唯一 ID。

`"host" : hostname`

这个键指定服务器主机。

现在查看日志看看哪一台被选为活跃节点。

再连接一下别的机器，查询一下命名空间 local.system.replset，会发现配置会在服务器间相互传递。



撰写本书时，副本集还在开发中，还没有进入 MongoDB 的生产版本。因此，这里的信息难免会有变化。关于副本集最新的文档详见 MongoDB wiki（http://www. mongodb.org/display/DOCS/Replica+Sets）。

9.2.2 副本集中的节点

任何时间，集群只有一个活跃节点，其他的都为备份节点。活跃节点实际上是活跃服务器，这里的不同是，指定的活跃节点可以随时间而改变。

有几种不同类型的节点可以存在于副本集中。

- `standard`

  这种就是常规节点，它存储一份完整的数据副本，参与选举投票，有可能成为活跃节点。

- `passive`

  存储了完整的数据副本，参与投票，不能成为活跃节点。

- `arbiter`

  仲裁者只参与投票，不接收复制的数据，也不能成为活跃节点。

标准节点和被动节点之间的区别仅仅就是数量的差别；每个参与节点（非仲裁者）有个优先权。优先权为 0 则是被动的，不能成为活跃节点。优先值不为 0，则按照由大到小选出活跃节点，优先值一样的话则看谁的数据比较新。所以，要是有两个优先值为 1 和一个优先值为 0.5 的节点，最后一个节点只有在前两个节点都不可用的时候才能成为活跃节点。

在节点配置中修改 priority 键，来配置成标准节点或者被动节点。

```
> members.push({
... "_id" : 3,
... "host" : "morton:10003",
... "priority" : 40
... });
```

默认优先级为 1，可以是 0 ～ 1 000（含）。

"arbiterOnly" 键可以指定仲裁节点。

```
> members.push({
... "_id" : 4,
... "host" : "morton:10004",
... "arbiterOnly" : true
... });
```

下一节将进一步介绍仲裁节点。

备份节点会从活跃节点抽取 oplog，并执行操作，就像活跃备份系统中的备份服务器一样。活跃节点也会写操作到自己的本地 oplog，这样就能成为活跃节点了。oplog 中的操作也包括严格递增的序号。通过这个序号来判定数据的时效性。

9.2.3　故障切换和活跃节点选举

如果活跃节点坏了，其余节点会选一个新的活跃节点出来。选举过程可以由任何非活跃节点发起。新的活跃节点由副本集中的大多数选举产生。仲裁节点也会参与投票，避免出现僵局（比如，当网络分割，参与节点被分成两半时）。新的活跃节点将是优先级最高的节点，优先级相同则数据较新的节点获胜（详见图 9-6 ～图 9-8）。

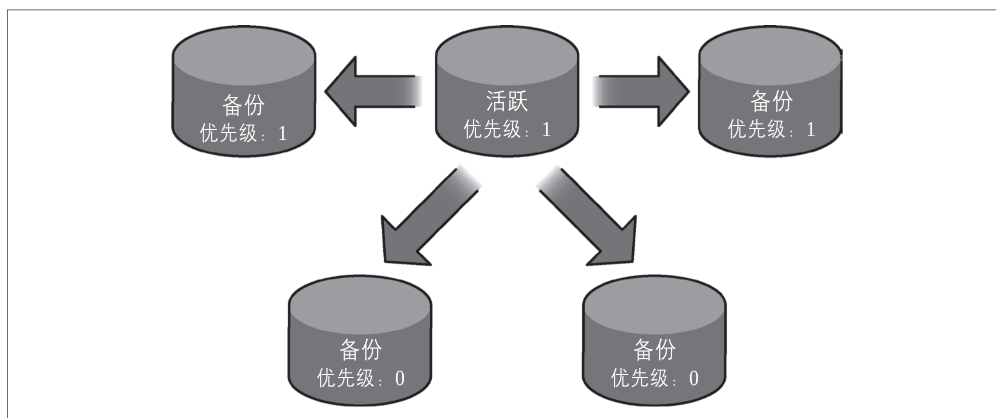


图 9-6：副本集有多个不同优先级的服务器

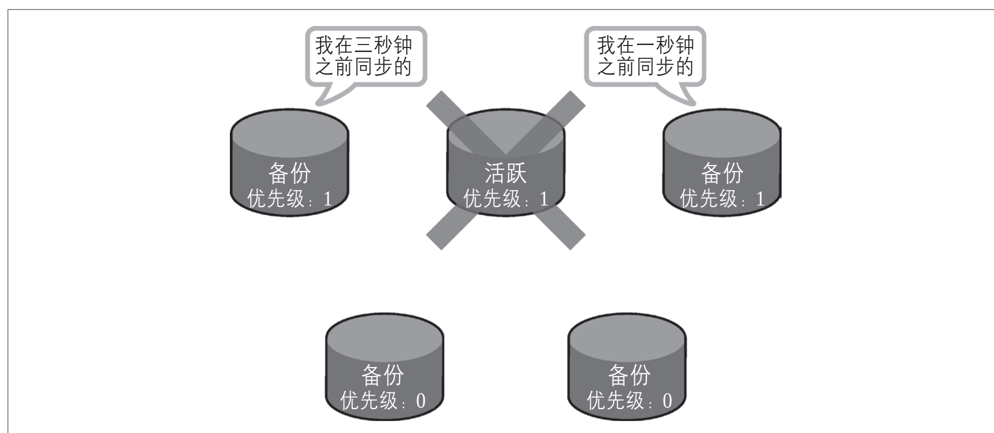


图 9-7：活跃节点坏了，具有最高优先级的服务器会比较数据的新旧程度

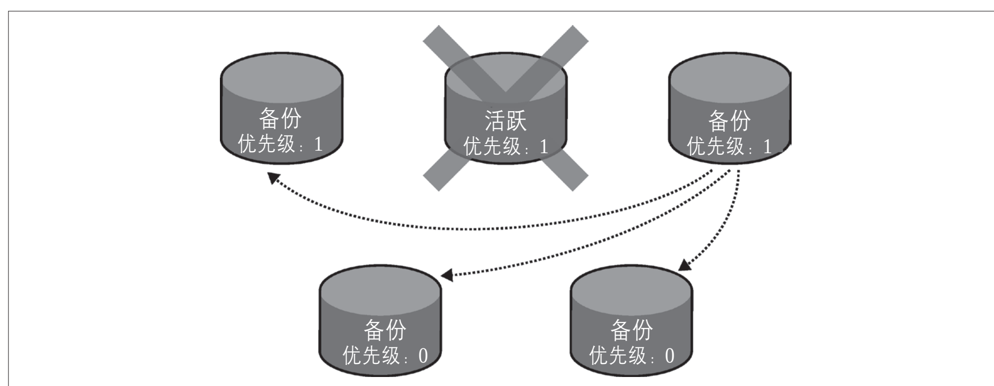


图 9-8：优先级最高且数据最新的服务器成为活跃节点

活跃节点使用心跳来跟踪集群中有多少节点对其可见。如果不够半数，活跃节点会自动降为备份节点。这样就能防止活跃节点一直不放权，比如当网络分割后已经与集群隔离开的时候。

不论活跃节点何时变化，新活跃节点的数据就被假定为系统的最新数据。对其他节点（即原来的活跃节点）的操作都会回滚，即便是之前的活跃节点已经恢复工作了。为了完成回滚，所有节点连接新的活跃节点后要重新同步。这些节点会查看自己的 oplog，找出其中活跃节点没有执行过的操作，然后向活跃节点请求这些操作影响的文档的最新副本。正在执行重新同步的节点被视为*恢复中*，在完成这个过程之前不能成为活跃节点候选者。

9.3　在从服务器上执行操作

从节点的主要作用是作为故障恢复机制，以防主节点数据丢失或者停止服务。但是

还有些别的用法。从节点可用做备份的数据源（详见第 8 章）。也可以用来扩展读取性能，或是进行数据处理。

9.3.1　读扩展

用 MongoDB 扩展读取的一种方式就是将查询放在从节点上。这样，主节点的负载就减轻了。一般说来，当负载是读取密集型时这是非常不错的方案。要是写入密集型，则要参见第 10 章，了解怎样用自动分片来进行扩展。

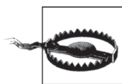


使用从节点来扩展 MongoDB 的读取有个要点，就是数据复制并不同步。也就是说在主节点插入和更新数据后，有片刻从节点的数据不是最新的。在考虑用查询从节点完成请求时这点非常重要。

扩展读取本身很简单：还像往常一样设置主从复制，连接从服务器处理请求。唯一的技巧就是有个特殊的查询选项，告诉从服务器是否可以处理请求。（默认是不可以的。）这个选项叫做 `slaveOkay`，所有的 MongoDB 驱动程序都提供了一种机制来设置它。有些驱动程序还提供工具使得将请求分布到从节点的过程自动化，但这个过程随驱动程序的不同而不同。

9.3.2　用从节点做数据处理

从节点的另外一个用途就是作为一种机制来减轻密集型处理的负载，或作为聚合，避免影响主节点的性能。用 `--master` 参数启动一个普通的从节点。同时使用 `--slave` 和 `--master` 有点矛盾。这意味着如果能对从节点进行写入、像往常一样查询，就把它作为一个普通的 MongoDB 主节点好了。从节点还是会不断地从真正的主节点复制数据。这样，就可以对从节点执行阻塞操作而不影响主节点的性能。



用这种技术的时候，一定要确保不能对正在复制主节点数据的从节点上的数据库执行写入。从节点不能恢复这些操作，就不能正确地映射主节点。

从节点第一次启动时也不能有正被复制的数据库。要是有的话，这个数据库就不能完成同步了，只能更新新的操作。

9.4　工作原理

总的说来，MongoDB 的复制至少需要两个服务器或者节点。其中一个是主节点，负责处理客户端请求，其他的都是从节点，负责映射主节点的数据。主节点记录在其上执行的所有操作。从节点定期轮询主节点获得这些操作，然后对自己的数据副本执行这些操作。由于和主节点执行了相同的操作，从节点就能保持与主节点的数据同步。

9.4.1 oplog

主节点的操作记录称为oplog（operation log的简写）。oplog存储在一个特殊的数据库中，叫做local。oplog就在其中的`oplog.$main`集合里面。oplog中的每个文档都代表主节点上执行的一个操作。文档包含的键如下。

- `ts`
  操作的时间戳。时间戳是一种内部类型，用于跟踪操作执行的时间。由4字节的时间戳和4字节的递增计数器构成。
- `op`
  操作类型，只有1字节代码。（例如"i"代表插入）
- `ns`
  执行操作的命名空间（集合名）。
- `o`
  进一步指定要执行的操作的文档。对插入来说，就是要插入的文档。

需要重点强调的是oplog只记录改变数据库状态的操作。比如，查询就不再存储在oplog中。这是因为oplog只是作为从节点与主节点保持数据同步的机制。

存储在oplog中的操作也不是完全和主节点的操作一模一样的。这些操作在存储之前先要做等幂变换，也就是说，这些操作可以在从服务器端多次执行，只要顺序是对的，就不会有问题。例如，使用`"$inc"`执行的增加更新操作，会被转换成`"$set"`操作。

另外一点要注意的就是，oplog存储在固定集合中（详见7.2节）。由于新操作也会存储在oplog里，它们会自动替换旧的操作。这样就能保证oplog不超过预先设定的大小。启动服务器时可以用`--oplogSize`指定这个大小，单位是MB。默认情况下，64位的实例将使用oplog 5%的可用空间。这个空间将在local数据库中分配，并在服务器启动时预先分配。

9.4.2 同步

从节点第一次启动时，会对主节点数据进行完整的同步。从节点复制主节点上的每个文档，耗费的资源可想而知。同步完成后，从节点开始查询主节点的oplog并执行这些操作，以保证数据是最新的。

如果从节点的操作已经被主节点落下很远了，从节点就跟不上同步了。跟不上同步的从节点无法一直不断地追赶主节点，因为主节点oplog的所有操作都太"新"了。从节点发生了宕机或者疲于应付读取时就会出现这种情况。也会在执行完完整同步以后发生类似的事，因为只要同步时间太长，同步完成时，oplog可能已经滚了一圈了。

从节点跟不上同步时，复制就会停下，从节点需要重新做完整的同步。可以用 {"resync" : 1} 命令手动执行重新同步，也可以在启动从节点时使用 --autoresync 选项让其自动重新同步。重新同步代价高昂，所以要尽量避免，方法就是配置足够大的 oplog。

为了避免从节点跟不上，一定要确保主节点的 oplog 足够大，能存放相当长时间的操作记录。大的 oplog 显然会占用更多的磁盘空间，这就需要权衡一下，找个折中点（默认的 oplog 大小是剩余磁盘空间的 5%）。关于 oplog 大小的相关事宜，参见 9.5 节。

9.4.3 复制状态和本地数据库

本地数据库用来存放所有内部复制状态，主节点和从节点都有。本地数据库的名字就是 local，其内容不会被复制。这样就能确保一个 MongoDB 服务器只有一个本地数据库。



本地数据库不限于存放 MongoDB 的内部状态。如果有不想被复制的文档，也可以将其放在本地数据库的集合里面。

主节点上的复制状态还包括从节点的列表（从节点连接主节点时会执行 handshake 命令进行握手）。这个列表存放在 slaves 集合中：

```
> db.slaves.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567c"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" :
1 } }
{ "_id" : ObjectId("4c128730e6e5c3096f40e0de"), "host" : "127.0.0.1",
  "ns" : "local.oplog.$main", "syncedTo" : { "t" : 1276282710000, "i" :
1 } }
```

从节点也在本地数据库中存放状态。在 me 集合中存放从节点的唯一标识符，在 sources 集合中存放源或节点的列表。

```
> db.sources.find()
{ "_id" : ObjectId("4c1287178e00e93d1858567b"), "host" :
"localhost:27017",
  "source" : "main", "syncedTo" : { "t" : 1276283096000, "i" : 1 },
  "localLogTs" : { "t" : 0, "i" : 0 } }
```

主节点和从节点都跟踪从节点的更新状况，这是通过存放在 "syncedTo" 中的时间戳来完成的。每次从节点查询主节点的 oplog 时，都会用 "syncedTo" 来确定哪些操作需要执行，或者查看是否已经跟不上同步了。

9.4.4 阻塞复制

开发者可以用 getLastError 的 "w" 参数来确保数据的同步性。这里运行 getLast-

Error 会进入阻塞状态，直到 *N* 个服务器复制了最新的写入操作为止。

```
> db.runCommand({getLastError: 1, w: N});
```

如果没有 *N*，或者小于 2，命令就会立刻返回。如果 *N* 等于 2，主节点要等到至少一个从节点复制了上个操作才会响应命令（主节点本身也包括在 *N* 里面）。主节点使用 local.slaves 中存放的 "syncedTo" 信息跟踪从节点的更新情况。

当指定 "w" 选项后，还可以使用 "wtimeout" 选项，表示以毫秒为单位的超时。getLastError 就能在上一个操作复制到 *N* 个节点超时时返回错误（默认情况下命令是没有超时的）。

阻塞复制会导致写操作明显变慢，尤其是 "w" 的值比较大时。实际上，对重要操作将其值设为 2 或者 3 就能效率与安全兼备了。

9.5 管理

这节会介绍复制管理的一些概念。

9.5.1 诊断

MongoDB 包含很多有用的管理工具，用以查看复制的状态。当连接到主节点后，使用 db.printReplicationInfo 函数：

```
> db.printReplicationInfo();
    configured oplog size:   10.48576MB
    log length start to end: 34secs (0.01hrs)
    oplog first event time:  Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
    oplog last event time:   Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
    now:                     Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

这些信息是 oplog 的大小和 oplog 中操作的时间范围。例子中的 oplog 大约是 10 MB，仅能放置大约 30 秒的操作。差不多是时候为 oplog 扩容了（详见下节）。oplog 的长度至少要能满足一次完整的重新同步。不然，从节点同步（或者重新同步）完成后发现已经跟不上了。



日志的长度是通过 oplog 中最早的操作时间和最后的操作时间的差值得到的。如果刚刚启动服务器，最早的操作会相对较新。这时，日志的长度就会很小，即便 oplog 可能还有空闲空间也是如此。所以说，当服务器跑了一段时间，日志已经转了个来回，这时日志长度才能准确度量记录的时间。

当连接到从节点时，用 db.printSlaveReplicationInfo() 函数，能得到从节点

的一些信息：

```
> db.printSlaveReplicationInfo();
    source:   localhost:27017
    syncedTo: Tue Mar 30 2010 16:44:01 GMT-0400 (EDT)
    = 12secs ago (0hrs)
```

显示的是从节点的数据源列表，其中有数据滞后时间。本例中只滞后 12 秒。

9.5.2 变更oplog的大小

若已经发现 oplog 大小不合适，最简单的做法就是停掉主节点，删除 local 数据库的文件，用新的设置（`--oplogSize`）重新启动。过程如下：

```
$ rm /data/db/local.*
$ ./mongod --master --oplogSize size
size is specified in megabytes.
```



为大型的 oplog 预分配空间非常耗费时间，且可能导致主节点停机时间增加，所以尽可能手动预分配数据文件。关于停止复制（http://www.mongodb.org/display/DOCS/Halted+Replication），详见 MongoDB 帮助文档。

重启主节点之后，所有从节点得用 `--autoresync` 重启，否则需要手动重新同步。

9.5.3 复制的认证问题

如果在复制中使用了认证（详见 8.3.1 节），还需要做些配置，使得从节点能够访问主节点的数据。在主节点和从节点上都需要在本地数据库添加用户，每个节点的用户名和口令都是相同的。本地数据库的用户类似 admin 中的用户，能够读写整个服务器。

从节点连接主节点时，会用存储在 *local.system.users* 中的用户进行认证。最先尝试“repl”用户，若没有此用户，则用 *local.system.users* 中的第一个可用用户。所以，按照如下步骤配置主节点和从节点，用可靠的密码替换 *password*，就能配置认证复制了：

```
> use local
switched to db local
> db.add User("repl", password);
{
    "user" : "repl",
    "readOnly" : false,
    "pwd" : "..."
}
```

从节点之后就可以复制主节点了。

第 10 章 分片

分片是 MongoDB 的扩展方式。通过分片能够增加更多的机器来应对不断增加的负载和数据，还不影响应用。

10.1 分片简介

分片（sharding）是指将数据拆分，将其分散存在不同的机器上的过程。有时也用*分区*（partitioning）来表示这个概念。将数据分散到不同的机器上，不需要功能强大的大型计算机就可以储存更多的数据，处理更大的负载。

使用几乎所有数据库软件都能进行*手动分片*。应用需要维护与若干不同数据库服务器的连接，每个连接还是完全独立的。应用程序管理不同服务器上的不同数据，存储查询都需要在正确的服务器上进行。这种方法可以很好地工作，但是非常难以维护，比如向集群添加节点或从集群删除节点都很困难，调整数据分布和负载模式也不轻松。

MongoDB 支持*自动分片*，可以摆脱手动分片的管理困扰。集群自动切分数据，做负载均衡。在本书的剩余部分里（以及差不多其他所有 MongoDB 的文档中），*分片*和*自动分片*是混用的，意思是一样的，但是要清楚应用中自动分片和手动分片的差别。

10.2 MongoDB中的自动分片

MongoDB 分片的基本思想就是将集合切分成小块。这些块分散到若干*片*里面，每个片只负责总数据的一部分。应用程序不必知道哪片对应哪些数据，甚至不需要知道数据已经被拆分了，所以在分片之前要运行一个路由进程，该进程名为 `mongos`。这个路由器知道所有数据的存放位置，所以应用可以连接它来正常发送请求。对应

用来说，它仅知道连接了一个普通的 mongod。路由器知道数据和片的对应关系，能够转发请求到正确的片上。如果请求有了回应，路由器将其收集起来回送给应用。

在没有分片的时候，客户端连接 mongod 进程，如图 10-1 所示。分片时客户端会连接 mongos 进程，如图 10-2 所示。mongos 对应用隐藏了分片的细节。从应用角度看，分片不分片没什么差别。所以需要扩展的时候，不必修改应用程序的代码。

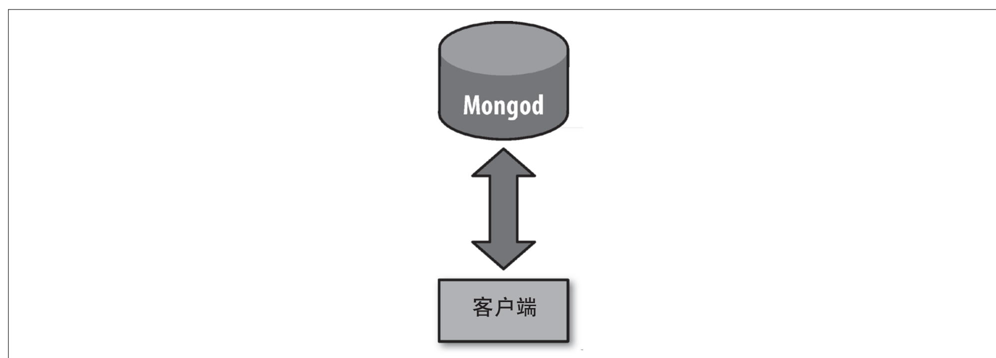


图 10-1：不分片的客户端连接

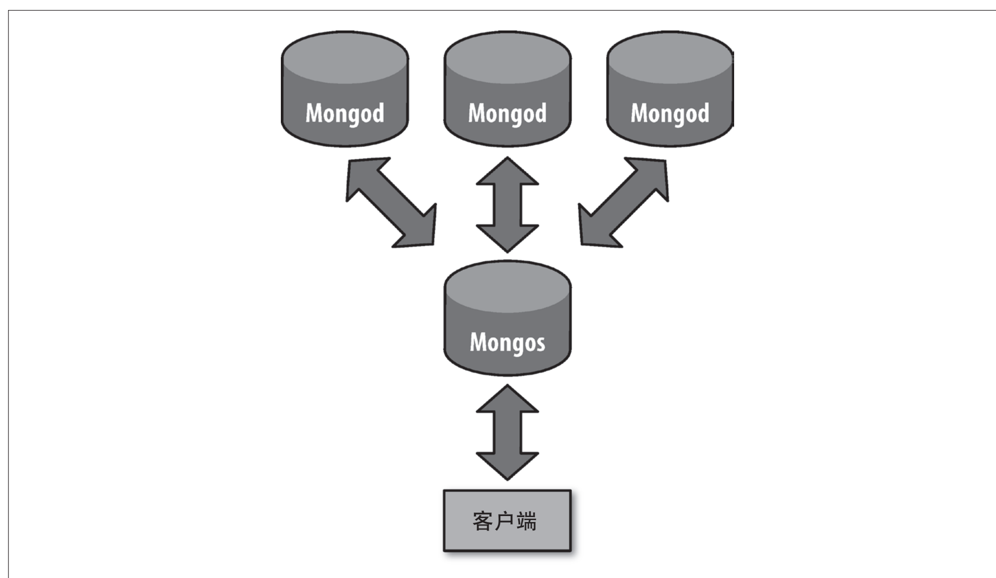


图 10-2：分片的客户端连接



如果现在还用的是老版本的 MongoDB，最好升级到 1.6.0 以上版本，之后再用分片。分片虽存在了一段时间了，但是从 1.6.0 后才是生产版本。

何时分片

常见问题就是什么时间开始分片呢。出现下面的信号时，就要考虑使用分片了。

- 机器的磁盘不够用了。
- 单个 `mongod` 已经不能满足写数据的性能需要了。
- 想将大量数据放在内存中提高性能。

一般说来，先要从不分片开始，然后在需要时将其转换成分片的。

10.3 片键

设置分片时，需要从集合里面选一个键，用该键的值作为数据拆分的依据。这个键称为*片键*（shard key）。

用个例子来说明这个过程：假设有个文档集合表示的是人员。如果选择名字（`"name"`）作为片键，第一片可能会存放名字以 A~F 开头的文档，第二片存的 G~P 的名字，第三片存的 Q~Z 的名字。随着添加（或者删除）片，MongoDB 会重新平衡数据，使每片的流量都比较均衡，数据量也在合理范围内（例如，流量比较大的片存放的数据或许会比流量小的片数据量要少些）。

10.3.1 将已有的集合分片

假设有个存储日志的集合，现在要分片。我们开启分片功能，然后告诉 MongoDB 用 `"timestamp"` 作为片键，就把所有数据放到了一个片上。可以随意插入数据，但总会是在一个片上。

而后，增加一个片。这个片建好并运行了以后，MongoDB 就会把集合拆分成两半，称为*块*。每个块中包括片键值在一定范围内的所有文档，所以就假设其中一块包含时间戳在 2003 年 6 月 26 日以前的文档，另一块含有 2003 年 6 月 27 日以后的文档。其中一块会被移动到新片上。

如果新文档的时间戳在 2003 年 6 月 27 日之前，则添加到第一个块，否则就加到另一个块。

10.3.2 递增片键还是随机片键

片键的选择决定了插入操作在片之间的分布。

如果选择了像 `"timestamp"` 这样的键，这个值很可能不断增长，而且没有太大的间断，就会将所有数据发送到一个片上（含有 [2003 年 6 月 27 日， ∞] 的那片）。

要知道，如果又添加了新片，再拆分数据，还是会都插入到一台服务器上。添加了新片，MongoDB 可能会将 [2003 年 6 月 27 日，∞] 拆分成 [2003 年 6 月 27 日，2010 年 12 月 12 日] 和 [2010 年 12 月 12 日，∞]。最后还是只用“至此以后”的那个片来插入。这就不适合写入负载很高的情况了，但按照片键查询会非常高效。

如果写入负载比较高，想均匀分散负载到各个片，就得选择分布均匀的片键。日志的例子中时间戳的散列值或者没有特定模式的 `"logMessage"` 都是符合这个条件的。

不论片键随机跳跃还是稳定增加，片键的变化至关重要。例如，如果有个 `"logLevel"` 键只有 3 种值 `"DEBUG"`、`"WARN"` 或 `"ERROR"`，MongoDB 就无论如何也不能把它作为片键将数据分成多于 3 块（因为只有 3 个值）。如果键的变化太少，但又想让其作为片键，可以将这个键与一个变化较大的键组合起来，创建一个复合片键，例如 `"logLevel"` 和 `"timestamp"` 组合。

选择片键并创建片键很像建索引，因为二者原理类似。事实上，片键也是最常用的索引。

10.3.3 片键对操作的影响

最终用户应该无法区分是否分片。然而，了解在选择不同片键的情况下查询有何不同，还是很有帮助的，尤其是使用了分片的时候。

假设还是那个上一节所说的集合，按照 `"name"` 分片，有 3 个片，其名字首字母的范围是 A~Z。下面会以不同的方式执行不同的查询：

```
db.people.find({"name" : "Susan"})
```

mongos 会将这个查询直接发送给 Q~Z 片，获得响应后，直接转发给客户端。

```
db.people.find({"name" : {"$lt" : "L"}})
```

mongos 会将其先后发送给 A~F 和 G~P 片。然后将结果转发给客户端。

```
db.people.find().sort({"email" : 1})
```

mongos 会在所有片上查询，返回结果时还会做归并排序，确保结果顺序正确。mongos 用游标从各个服务器上获取数据，所以不必等到全部数据都拿到才向客户端发送批量结果。

```
db.people.find({"email" : "joe@example.com"})
```

mongos 并不跟踪 `"email"` 键，所以也不知道应该将查询发给哪个片。所以，它就向所有片顺序发送查询。

如果插入文档的话，mongos 会依据 `"name"` 键的值，将其发送到相应的片上。

10.4 建立分片

建立分片有两步：启动实际的服务器，然后决定怎么切分数据。

分片一般会有 3 个组成部分。

- 片
  片就是保存子集合数据的容器。片可是单个的 mongod 服务器（开发和测试用），也可以是副本集（生产用）。所以，即便一片内有多台服务器，也只能有一个主服务器，其他的服务器保存相同的数据。

- mongos
  mongos 就是 MongoDB 各版本中都配的路由器进程。它路由所有请求，然后将结果聚合。它本身并不存储数据或者配置信息（但会缓存配置服务器的信息）。

- 配置服务器
  配置服务器存储了集群的配置信息：数据和片的对应关系。mongos 不永久存放数据，所以需要个地方存放分片配置。它会从配置服务器获取同步数据。

如果已经能用了 MongoDB，就有了一个整装待发的片了（当前的 mongod 可以变成第一个片）。下一节会介绍如何从头建立一个片，但这不是表示一定要这么做，从已有的数据库开始是没问题的。

10.4.1 启动服务器

首先要启动配置服务器和 mongos。配置服务器需要最先启动，因为 mongos 会用到其上的配置信息。配置服务器的启动就像普通 mongod 一样。

```
$ mkdir -p ~/dbs/config
$ ./mongod --dbpath ~/dbs/config --port 20000
```

配置服务器不需要很多空间和资源（200 MB 实际数据大约占用 1 KB 的配置空间）。

现在就可以建立 mongos 进程，以供应用程序连接。这种路由服务器连数据目录都不需要，但一定要指明配置服务器的位置：

```
$ ./mongos --port 30000 --configdb localhost:20000
```

分片管理通常是通过 mongos 完成的。

添加片

片就是普通的 mongod 实例（或者副本集）：

```
$ mkdir -p ~/dbs/shard1
```

```
$ ./mongod --dbpath ~/dbs/shard1 --port 10000
```

现在连接刚才启动的 mongos，为集群添加一个片。启动 shell，连接 mongos：

```
$ ./mongo localhost:30000/admin
MongoDB shell version: 1.6.0
url: localhost:30000/admin
connecting to localhost:30000/admin
type "help" for help
>
```

确定连接的是 mongos 而不是 mongod 后，就可以通过 addshard 命令添加片了：

```
> db.runCommand({addshard : "localhost:10000", allowLocal : true})
{
    "added" : "localhost:10000",
    "ok" : true
}
```

当在 localhost 上运行片时，得设定 "allowLocal" 键。MongoDB 尽量避免由于错误配置，将集群配置到本地，所以得让它知道这仅仅是开发，而且我们很清楚自己在做什么。如果是在生产环境中，则要将其部署在不同的机器上（虽然可能会有交叠，详见下节）。

想添加片的时候，就运行 addshard。MongoDB 会负责将片集成到集群。

10.4.2　切分数据

MongoDB 不会将存储的每一条数据都直接发布，得先在数据库和集合的级别将分片功能打开。下面的例子欲以 "_id" 为基准切分 foo 数据库的 bar 集合。首先得开启 foo 的分片功能：

```
> db.runCommand({"enablesharding" : "foo"})
```

对数据库分片后，其内部的集合便会存储到不同的片上，同时也是对这些集合分片的前置条件。

在数据库的级别启用了分片以后，就可以使用 shardcollection 命令来对集合进行分片了：

```
> db.runCommand({"shardcollection" : "foo.bar", "key" : {"_id" : 1}})
```

这样集合就按照 "_id" 分片了。再添加数据，就会依据 "_id" 的值自动分散到各个片上。

10.5　生产配置

前一节的例子对于尝试分片开发而言还是不错的。但应用进入产品环境以后，就需

要更加健壮的方案了。成功地构建分片需要如下条件。

- 多个配置服务器。
- 多个 mongos 服务器。
- 每个片都是副本集。
- 正确设置 w。（有关 w 和复制的更多信息，详见上一章。）

10.5.1 健壮的配置

建立多个配置服务器是非常简单的。本书写作时，可以建立一个配置服务器（开发用）或者三个配置服务器（生产用）。

设置多个配置服务器和设置一个完全一样，就是重复 3 次而已：

```
$ mkdir -p ~/dbs/config1 ~/dbs/config2 ~/dbs/config3
$ ./mongod --dbpath ~/dbs/config1 --port 20001
$ ./mongod --dbpath ~/dbs/config2 --port 20002
$ ./mongod --dbpath ~/dbs/config3 --port 20003
```

然后，启动 mongos 的时候应将其连接到这 3 个配置服务器：

```
$ ./mongos --configdb localhost:20001,localhost:20002,localhost:20003
```

配置服务器使用的是两步提交机制，而不是普通 MongoDB 的异步复制，来维护集群配置的不同副本。这样能保证集群状态的一致性。这也意味着，某台配置服务器宕掉了以后，集群配置信息将是只读的。客户端还是能够读写，但是只有所有配置服务器备份了以后才能重新均衡数据。

10.5.2 多个mongos

mongos 的数量不受限制。建议针对一个应用服务器只运行一个 mongos 进程。这样每个应用服务器就可以与 mongos 进行本地会话，如果服务器不工作了，就不会有应用试图与不在的 mongos 通话了。

10.5.3 健壮的片

生产环境中，每个片都应是副本集。这样单个的服务器坏了，就不会导致整个片失效。用 addshard 命令就可以将副本集作为片添加。添加时只要指定副本集的名字和种子就好了。

比如要添加副本集 foo，其中包含一个服务器 prod.example.com:27017（还有别的服务器），就可以使用下列命令将其添加到集群里：

```
> db.runCommand({"addshard" : "foo/prod.example.com:27017"})
```

如果 prod.example.com 挂了，mongos 会知道它所连接的是一个副本集，并会使用新的主节点。

10.5.4　物理服务器

貌似需要太多机器了：3 个配置服务器，每片最少两个 mongod，还有若干 mongos 进程。但是这些不一定都需要单独的服务器。重要的是不把所有鸡蛋放在一个篮子里。好比不把所有 3 个配置服务器放到一台机器上，不把所有 mongos 放到一台机器上，不把这个副本集放到一台机器上。但是可以把一个配置服务器、一些 mongos 进程和副本集的一个节点放到一台机器上。

10.6　管理分片

分片信息主要存放在 config 数据库上，这样就能被任何连接到 mongos 的进程访问到了。

10.6.1　配置集合

下几节的代码都假设已经在 shell 中连接了 mongos，并且已经运行了 use config。

1. 片

可以在 shards 集合中查到所有的片：

```
> db.shards.find()
{ "_id" : "shard0", "host" : "localhost:10000" }
{ "_id" : "shard1", "host" : "localhost:10001" }
```

每片都有一个唯一的、好认的 _id。

2. 数据库

databases 集合含有已经在片上的数据库列表和一些相关信息：

```
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
{ "_id" : "x", "partitioned" : false, "primary" : "shard0" }
{
    "_id" : "test",
    "partitioned" : true,
    "primary" : "shard0",
    "sharded" : {
        "test.foo" : {
            "key" : {"x" : 1},
            "unique" : false
        }
    }
}
```

这里是全部可用的数据库和一些基本信息。

- "_id"，字符串

  "_id" 表示数据名。

- "partitioned"，布尔型

  如果为 true，则表示已经启用分片功能。

- "primary"，字符串

  这个值与片的 "_id" 相对应，表明这个数据库的“大本营”在哪里。不论分片与否，数据库总是会有个大本营的。要是分片了的话，创建数据库时会随机选择一个片。也就是说，大本营是开始创建数据库文件的位置。虽然分片时数据库也会用到很多别的服务器，但会从这个片开始。

3. 块

块信息保存在 chunks 集合中。这有很多有趣的东西，也可以看到数据到底是怎么切分到集群的：

```
> db.chunks.find()
 {
    "_id" : "test.foo-x_MinKey",
    "lastmod" : { "t" : 1276636243000, "i" : 1 },
    "ns" : "test.foo",
    "min" : {
        "x" : { $minKey : 1 }
    },
    "max" : {
        "x" : { $maxKey : 1 }
    },
    "shard" : "shard0"
}
```

单块的集合就是这样的：块的范围从 - ∞ （MinKey）到 ∞ （MaxKey）。

10.6.2　分片命令

前面已经介绍了一些基本命令，像添加块、启用分片。还有些命令对于管理集群也非常有帮助。

1. 获得概要

printShardingStatus 给出前面说的那些集合的概要：

```
> db.printShardingStatus()
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
```

```
        { "_id" : "shard0", "host" : "localhost:10000" }
        { "_id" : "shard1", "host" : "localhost:10001" }
    databases:
        { "_id" : "admin", "partitioned" : false, "primary" : "config" }
        { "_id" : "foo", "partitioned" : false, "primary" : "shard1" }
        { "_id" : "x", "partitioned" : false, "primary" : "shard0" }
        { "_id" : "test", "partitioned" : true, "primary" : "shard0",
          "sharded" : { "test.foo" : { "key" : { "x" : 1 }, "unique" : false }
  } }
              test.foo chunks:
                  { "x" : { $minKey : 1} } -->> { "x" : { $maxKey : 1} } on : shard0
                  { "t" : 1276636243000, "i" : 1 }
```

2. 删除片

用 removeshard 就能从集群中删除片。removeshard 会把给定片上的所有块都挪到其他片上。

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "started draining" : "localhost:10000",
    "ok" : 1
}
```

在挪动过程中，removeshard 会显示进程：

```
> db.runCommand({"removeshard" : "localhost:10000"});
{
    "msg" : "already draining...",
    "remaining" : {
        "chunks" : 39,
        "dbs" : 2
    },
    "ok" : 1
}
```

最后，挪动完毕，removeshard 会提示片已被成功删除。



在 1.6.0 版本中，如果删除的片是数据库的大本营（基片），必须手动移动数据库（使用 moveprimary 命令）：

```
> db.runCommand({"moveprimary" : "test", "to" :
"localhost:10001"})
{
    "primary" : "localhost:10001",
    "ok" : 1
}
```

以后的版本会将其自动化。

第 11 章 应用举例

本书前面所举的例子都是 JavaScript 的。本章将探索实际应用中更常用的语言是如何与 MongoDB 配合的。

11.1 化学品搜索引擎：Java

Java 驱动程序是 MongoDB 最早的驱动。它已经用于生产环境多年，而且非常稳定，是企业级开发的首选。

下面会使用 Java 驱动程序构建一个化合物的搜索引擎。这个例子受到 http://www.chemeo.com 网站的启发。这个搜索引擎有成千上万的化合物的化学性质和物理性质，其目标就是使这些信息全部能够被搜索到。

11.1.1 安装Java驱动程序

Java 驱动程序是一个 JAR 文件，可以从 Github（http://www.github.com/mongodb/mongojava-driver/downloads）上下载。将这个 JAR 加入到类路径中完成安装。

一般应用需要使用的所有 Java 类都在 `com.mongodb` 和 `com.mongodb.gridfs` 两个包中。.JAR 中还有些其他的包，当想要修改驱动内部结构或是扩展功能时会用得到，但绝大部分应用都不会用到。

11.1.2 使用Java驱动程序

就和 Java 里的大部分东西一样，这个 API 有点啰嗦（尤其是与其他语言的 API 相比时）。然而，所有的概念都类似于使用 shell，几乎所有的方法名都是完全相同的。

com.mongodb.Mongo 类会与 MongoDB 服务器建立连接。之后就可以通过集合访问数据库，然后获得数据库的连接：

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;

class ChemSearch {

    public static void main(String[] args) {
        Mongo connection = new Mongo();
        DB db = connection.getDB("search");
        DBCollection chemicals = db.getCollection("chemicals");

        /* ... */
    }
}
```

这样连接的是 localhost:27017，并得到命名空间 search.chemicals。

Java 中的文档必须是 org.bson.DBObject 的实例，org.bson.DBObject 是一个接口，可认为是一个有序的 java.util.Map。在 Java 中有多种创建文档的方式，最简单的要数用 com.mongodb.BasicDBObject 类了。因此，创建能由 shell 表示为 {"x" : 1, "y" : "foo"} 的这种文档操作如下：

```
BasicDBObject doc = new BasicDBObject();
doc.put("x", 1);
doc.put("y", "foo");
```

想要添加内嵌文档，如 "z" : {"hello" : "world"}，就得先创建另外一个 BasicDBObject 对象，然后将其放入到上一级：

```
BasicDBObject z = new BasicDBObject();
z.put("hello", "world");

doc.put("z", z);
```

这样就有了文档 {"x" : 1, "y" : "foo", "z" : {"hello" : "world"}}。

所有 Java 驱动程序实现的其他方法都和 shell 中的类似，例如可以用 chemicals.insert(doc) 或者 chemicals.find(doc)。完整的 Java 驱动程序的 API 文档位于 http://api.mongodb.org/java，MongoDB Java 语言中心（http://www.mongodb.org/display/DOCS/Java+Language+Center）还有些文章介绍了另外几个方面的话题（并行性、数据类型等）。

11.1.3 模式设计

这个问题的难点在于每种化学物质都有数以千计的属性，而且要能对全部这些属性

做快速的搜索。举两个简单的例子：硅和氮化硅。硅的文档比较简单：

```
{
    "name" : "silicon",
    "mw" : 32.1173
}
```

mw 表示“分子量”。

氮化硅就会有很多属性，所以它的文档要复杂一点：

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH° gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S° gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    }
}
```

MognoDB 能存放任意数量、任意属性结构的化学物质，这样应用就能轻易扩展，但目前还不能对当前的格式进行有效的索引。想要快速搜索属性，就得对几乎所有键进行索引！基于第 5 章所学，就能判断这绝不是一个好主意。

不过还是有办法的。MongoDB 索引会将数组的每一个元素都涵盖进去，利用这一特点，可以将想要索引的属性放到一个包含常用键名的数组中。例如，对于氮化硅，可以为索引添加一个数组，将全部属性放到数组中：

```
{
    "name" : "silicon nitride",
    "mw" : 42.0922,
    "ΔfH° gas" : {
        "value" : 372.38,
        "units" : "kJ/mol"
    },
    "S° gas" : {
        "value" : 216.81,
        "units" : "J/mol×K"
    },
    "index" : [
        {"name" : "mw", "value" : 42.0922},
        {"name" : "ΔfH° gas", "value" : 372.38},
        {"name" : "S° gas", "value" : 216.81}
    ]
}
```

对于硅，数组只有一个元素，就是分子量：

```
{
    "name" : "silicon",
    "mw" : 32.1173,
    "index" : [
        {"name" : "mw", "value" : 32.1173}
    ]
}
```

现在只要创建一个 "index.name" 和 "index.value" 的复合索引就可以了。这样就能很快地搜索化学品的任何特性了。

11.1.4 用Java实现

回到前面那段 Java 代码，现在使用 ensureIndex 函数建复合索引：

```
BasicDBObject index = new BasicDBObject();
index.put("index.name", 1);
index.put("index.value", 1);

chemicals.ensureIndex(index);
```

创建氮化硅的文档不难，但是很繁琐：

```
public static DBObject createSiliconNitride() {
    BasicDBObject sn = new BasicDBObject();
    sn.put("name", "silicon nitride");
    sn.put("mw", 42.0922);

    BasicDBObject deltafHgas = new BasicDBObject();
    deltafHgas.put("value", 372.38);
    deltafHgas.put("units", "kJ/mol");

    sn.put("ΔfH° gas", deltafHgas);

    BasicDBObject sgas = new BasicDBObject();
    sgas.put("value", 216.81);
    sgas.put("units", "J/mol×K");

    sn.put("S° gas", sgas);

    ArrayList<BasicDBObject> index = new ArrayList<BasicDBObject>();
    index.add(BasicDBObjectBuilder.start()
              .add("name", "mw").add("value", 42.0922).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "ΔfH° gas").add("value", 372.38).get());
    index.add(BasicDBObjectBuilder.start()
              .add("name", "S° gas").add("value", 216.81).get());

    sn.put("index", index);

    return sn;
}
```

只要实现了 java.util.List，就可以用来表示数组，所以用 java.util.ArrayList 来存放表示化学物质属性的内嵌文档。

11.1.5 一些问题

这种结构的一个问题就是，如果查询含有多个条件，条件的顺序就很重要。比如要查找分子量小于 1000、沸点大于 0°、凝固点是 -20° 的文档，最简单的做法就是用 "$all" 将所有查询条件都放进去：

```
BasicDBObject criteria = new BasicDBObject();

BasicDBObject all = new BasicDBObject();

BasicDBObject mw = new BasicDBObject("name", "mw");
mw.put("value", new BasicDBObject("$lt", 1000));

BasicDBObject bp = new BasicDBObject("name", "bp");
bp.put("value", new BasicDBObject("$gt", 0));

BasicDBObject fp = new BasicDBObject("name", "fp");
fp.put("value", -20);

all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
all.put("$elemMatch", fp);
criteria.put("index", new BasicDBObject("$all", all));

chemicals.find(criteria);
```

这样做的问题是，MongoDB 只用索引来辅助 "$all" 条件句的第一项。假设有一百万个文档的 "mw" 值小于 1000。MongoDB 可以利用索引来完成这部分查询，但是还会用扫描的方式查找沸点和凝固点，这就会花费很长的时间。

要是已知数据的一些特征，比如知道只有 43 种化学物质的凝固点为 -20°，这样就可以调整一下顺序：

```
all.put("$elemMatch", fp);
all.put("$elemMatch", mw);
all.put("$elemMatch", bp);
criteria.put("index", new BasicDBObject("$all", all));
```

这样数据库就会很快找到 43 个元素，后面的条件只需要扫描 43 个元素就好了（而不是一百万个元素）。当然，找出所有查询的合理顺序相当有难度，涉及模式识别和数据聚合算法，这些内容超出了本书的范畴。

11.2 新闻聚合器：PHP

接下来会创建一个简单的新闻聚合器应用：用户提交感兴趣的站点的链接，其他用

户可以评论，对链接质量发起投票（也包含他人的评论）。这需要建立一个评论树和实现一个投票系统。

11.2.1　安装PHP驱动程序

MongoDB 的 PHP 驱动程序是一个 PHP 扩展。在绝大部分平台下都很容易安装。PHP 5.1 及以上版本的系统就可以了。

1. 在Windows下安装

先要查看 `phpinfo()` 的输出，看看运行的 PHP 版本（在 Windows 下只支持 PHP 5.2 和 5.3，不支持 5.1），包括 VC 版本。如果用的是 Apache，则需要 VC6，否则需要 VC9。有些 Zend 用的是 VC8。还要注意是否是线程安全的（thread-save，通常缩写为“ts”）。

看 `phpinfo()` 的时候，留意 `extension_dir` 的值，一会儿就在那里安装扩展组件。

现在就轻车熟路了，转到 Github（http:www.github.com/mongodb/mongo-php-driver/dow- nloads），下载和 PHP 版本、VC 版本和线程安全相匹配的包。解压，将 php_mongo.dll 移动到 extension_dir 目录。

最后在 php.ini 文件中添加如下一行：

```
extension=php_mongo.dll
```

如果启动了应用服务器（Apache、WAMPP 等），请重启一下。下次启动 PHP 时会自动加载 Mongo 扩展。

2. 在Mac OS X下安装

要是有 PECL 的话，这是安装扩展组件的最简单方式了：

```
$ pecl install mongo
```

有些 Mac 没有自带 PECL 或者安装扩展组件的 PHP 库不正确。

要是 PECL 不能正常工作，可以下载为 OS X 编译好的二进制文件，地址在 Github（http://www.github.com/mongodb/mongo-php-driver/downloads）。执行 `php -i` 查看运行的 PHP 版本和 `extension_dir` 的值，然后下载合适的版本（文件名中含有“osx”）。解压缩，然后将 mongo.so 移动到 extension_dir 指定的目录。

安装完后，在 php.ini 文件添加下列命令：

```
extension=mongo.so
```

重启应用服务器，Mongo 扩展组件会在下次 PHP 启动时自动加载。

3. 在Linux和Unix下安装

运行如下命令：

```
$ pecl install mongo
```

在 php.ini 中添加下面一行：

```
extension=mongo.so
```

重启应用服务器，Mongo 扩展组件会在下次 PHP 启动时自动加载。

11.2.2 使用PHP驱动程序

Mongo 类就是一个到数据库的连接。默认情况下，构造器尝试连接本地默认端口处运行的数据库服务器。

用 __get 函数从连接中取得数据库，同样也可以从数据库取得集合（甚至从集合取得子集合）。例如，下面语句连接 MongoDB 后，获取了数据库 foo 的 bar 集合：

```
<?php

$connection = new Mongo();

$collection = $connection->foo->bar;

?>
```

还可以继续链接取值函数来访问子集合。例如要得到 bar.baz 集合，可以使用如下命令：

```
$collection = $connection->foo->bar->baz;
```

文档用 PHP 中的关联数组表示。这样，JavaScript 中的 {"foo" : "bar"} 就可以表示成 PHP 中的 array("foo" => "bar")，数组对应表示为 PHP 中的数组，这有时有点费解：JavaScript 的 ["foo", "bar", "baz"] 等价于 array("foo", "bar", "baz")。

对于 null、布尔型、数字型、字符串和数组，PHP 驱动使用了 PHP 的本机类型。对于其他类型，有一类以 Mongo 作为前缀，MongoCollection 是集合，MongoDB 是数据库，MongoRegex 是正则表达式。这些类在 PHP 手册（http://us2.php.net/manual/en/book.mongo.php）里有详尽的说明。

11.2.3 设计新闻聚集器

创建的新闻聚集器较为简单，用户可以提交有趣故事的链接，其他用户可以对此进

行评论或者投票。这里仅做两个点：创建评论树和处理投票。

存放提交和评论，一个集合就够用了，这个集合称作 posts。起初的 posts 链接着某篇文章，形式大致如下：

```
{
    "_id" : ObjectId(),
    "title" : "A Witty Title",
    "url" : "http://www.example.com",
    "date" : new Date(),
    "votes" : 0,
    "author" : {
        "name" : "joe",
        "_id" : ObjectId(),
    }
}
```

评论的形式也类似，只不过没有 "url"，而是 "content"。

11.2.4　评论树

MongoDB 中表示树的方法有多种，选择哪种表示方法取决于要执行的查询的类型。

这里每个节点都有个数组，包含父节点、祖父节点，等等。所以比如有下面的评论结构：

```
original link
|- comment 1
|    |- comment 3 (reply to comment 1)
|    |- comment 4 (reply to comment 1)
|        |- comment 5 (reply to comment 4)
|- comment 2
|    |- comment 6 (reply to comment 2)
```

评论 5 的祖先数组会包含原始链接的 _id、评论 1 的 _id、评论 4 的 _id。评论 6 的祖先则有原始链接的 _id 和评论 2 的 _id。这样非常便于搜索“链接 X 的所有评论”或者“评论 2 的回复的子树”。

用这样的方式存放评论基于如下假设：有很多评论，但只对某一小部分评论感兴趣。要是想显示所有评论，且总数不会达到好几千条，就可以将整个评论树作为提交的链接文档的内嵌文档了。

使用这种祖先数组的方式，当要添加新评论时，得向集合添加新文档。创建这个叶子文档，要将其与父节点的 "_id" 值和其祖先数组联系起来。

```
function createLeaf($parent, $replyInfo) {
    $child = array(
        "_id" => new MongoId(),
```

```
        "content" => $replyInfo['content'],
        "date" => new MongoDate(),
        "votes" => 0,
        "author" => array(
            "name" => $replyInfo['name'],
            "name" => $replyInfo['name'],
        ),
        "ancestors" => $parent['ancestors'],
        "parent" => $parent['_id']
    );

    // add the parent's _id to the ancestors array
    $child['ancestors'][] = $parent['_id'];

    return $child;
}
```

之后就可以为 posts 集合添加新评论了：

```
$comment = createLeaf($parent, $replyInfo);

$posts = $connection->news->posts;
$posts->insert($comment);
```

然后就查询一下最新的提交（不是评论）：

```
$cursor = $posts->find(array("ancestors" => array('$size' => 0)));
$cursor = $cursor->sort(array("date" => -1));
```

如果要查看指定文章的评论，使用如下命令就可以找到所有评论：

```
$cursor = $posts->find(array("ancestors" => $postId));
```

事实上，可以用这个查询来访问评论的所有子树。子树的根节点作为 $postId，每个子节点都会在其祖先数组中含有 $postId，这样就被返回了。

为了加快查询速度，需要按照 "date" 和 "ancestors" 建立索引：

```
$pageOfComments = $posts->ensureIndex(array("date" => -1, "ancestors"
                                                           => 1));
```

现在就能非常快速地查询主页面、评论树或评论子树了。

11.2.5 投票

投票也有多种实现方式，取决于需要的功能和信息：允许来来回回地投票吗？要不要阻止用户多次投票？允许改变选择吗？你关心大家投票的时间吗，以便查看链接的趋势？每个需求都有不同的解决方法，需要很复杂的代码，而最简单投票就是用 "$inc"：

```
$posts->update(array("_id" => $postId),array('$inc' => array("votes", 1)));
```

对于有争议的或是热门的连接，不能让大家可以投好几百次，所以要限制每个用户只能对一个链接投一次。最简单的做法是加一个 "voters" 数组，记录一下投票的人，也就是用个数组把用户的 "_id" 值都记录下来。如果有人投票，执行更新操作，检查一下用户的 "_id" 是否在 "_id" 数组中：

```
$posts->update(array("_id" => $postId, "voters" => array('$ne' => $userId)),
    array('$inc' => array("votes", 1), '$push' => array("voters" =>
           $userId)));
```

如果有几百万用户，这样做没有问题。要是投票数量再大点，就会达到 4 MB 的上限，这样就得对特别热门的链接特殊处理一下，将放不下的投票放到新的文档中。

11.3 自定义提交表单：Ruby

MongoDB 在 Ruby 开发者中大受欢迎，很可能是因为面向文档的方式与 Ruby 的动态灵活的风格相契合。这个例子将用 MongoDB 的 Ruby 驱动程序建立一个自定义提交表单的框架，这个例子是《纽约时报》的博客文章，介绍怎样用 MongoDB 处理提交表单（http://open.blogs.nytimes.com/2010/05/25/building-a-better-submission-form/）的简化版。要查看在 Ruby 中使用 MongoDB 更为详尽的说明，请参考 Ruby 语言中心（http://www.mongodb.org/display/DOCS/Ruby+Language+Center）。

11.3.1 安装Ruby驱动

Ruby 驱动是个 RubyGem，地址为 http://rubygems.org。使用 gem 来安装是最方便的方式了。事先要安装最新的 RubyGems（使用 gem update --system），然后安装 mongo gem:

```
$ gem install mongo
Successfully installed bson-1.0.2
Successfully installed mongo-1.0.2
2 gems installed
Installing ri documentation for bson-1.0.2...
Building YARD (yri) index for bson-1.0.2...
Installing ri documentation for mongo-1.0.2...
Building YARD (yri) index for mongo-1.0.2...
Installing RDoc documentation for bson-1.0.2...
Installing RDoc documentation for mongo-1.0.2...
```

mongo gem 依赖于 bson gem，所以会一同安装。bson gem 负责驱动的所有 BSON 编码和解码（关于 BSON，详见附录 C 中的“BSON”一节）。如果有 bson_ext 的话，bson 就会利用其中的 C 扩展来提高性能。出于性能考虑，如果安装了 gem 一般都要安装 bson_ext：

```
$ gem install bson_ext
Building native extensions.  This could take a while...
Successfully installed bson_ext-1.0.1
1 gem installed
```

只要 `bson_ext` 在加载路径下，就会自动被调用的。

11.3.2 使用Ruby驱动

可以用 `Mongo::Connection` 类连接 MongoDB 实例。生成了 `Mongo::Connection` 以后，用方括号就可以得到数据库（这里用到的就是 stuffy 数据库）：

```
> require 'rubygems'
 => true
> require 'mongo'
 => true
> db = Mongo::Connection.new["stuffy"]
```

Ruby 驱动用散列表示文档。除此之外，其 API 和 shell 的 API 非常相似，方法名基本相同（Ruby 用的是下划线命名法，shell 用的是驼峰式命名法）。下面的方式在 bar 集合中插入文档 `{"x" : 1}`，然后查询结果：

```
> db["bar"].insert :x => 1
 => BSON::ObjectID('4c168343e6fb1b106f000001')
> db["bar"].find_one
 => {"_id"=>BSON::ObjectID('4c168343e6fb1b106f000001'), "x"=>1}
```

Ruby 中的文档有些地方值得注意。

- Ruby 1.9 的散列是有序的，这与 MongoDB 中文档的工作方式相匹配。但在 Ruby 1.8 中，散列是无序的。如有必要，驱动提供了一种特殊的类型 `BSON::OrderedHash`，当键的顺序很重要时用其替代普通的散列，
- 准备存放到 MongoDB 中的散列中的键和值都可以是符号，但是从 MongoDB 中返回的散列中，原来的值是符号输出就是符号，但是原来的键若是符号则返回时变成字符串。所以 `{:x => :y}` 就变成 `{"x" => :y}` 了。这是用 BSON 表示文档的副作用（关于 BSON，详见附录 C）。

11.3.3 自定义表单提交

现在的问题是为用户提交的数据生成自定义的表单，用这些表单来处理用户提交的信息。编辑器创建的表单，可以含有各种各样的字段，每个字段可以有不同的类型，可以有不同校验规则。这里会利用嵌入文档，将每个字段都作为表格内部的一个独立的内嵌文档存储。一个评论提交表单的形式大概如下：

```
comment_form = {
    :_id => "comments",
    :fields => [
        {
            :name => "name",
```

```
                :label => "Your Name",
                :help_text => "Required",
                :required => true,
                :type => "string",
                :max_length => 200
            },
            {
                :name => "email",
                :label => "Your E-mail Address",
                :help_text => "Required, but will not be displayed",
                :required => true,
                :type => "email"
            },
            {
                :name => "comment",
                :label => "Your Comment",
                :help_text => "Comments will be moderated",
                :required => true,
                :type => "string",
                :word_limit => 200
            }
        ]
    }
```

这个表格就体现出像 MongoDB 这种面向文档的数据库的好处。首先，可以在表单文档中直接嵌入字段，而不必存到别的地方然后进行连接。现在可以用一个简单的查询得到表单的整个表示，还能对不同类型的字段使用不同的键。在上面的例子中，name 字段有 :max_length 键，comment 字段有 :word_limit 键，而 email 字段就没有这两个键。

"_id" 存放的是好认的表单名，因为后面还要按照表单名创建索引来加快查询速度。将索引设成唯一索引，还能保证表单名不重复。

编辑器创建新表格时，只需保存最后的结果文档。保存刚才创建的 comment_form（评论表格）文档：

```
db["forms"].save comment_form
```

每次想要呈现包含评论表格的页面时，要先用表单名将表单文档调出来：

```
db["forms"].find_one :_id => "comments"
```

返回的这一个文档就包含了呈现表单所需的全部信息了，包括名字、标签、要呈现的每个输入字段的类型。当表单发生变化时，编辑器可以很容易地添加新字段，或者为已有的字段添加新的约束。

当用户提交时，可以用之前用过的查询得到相关的表单文档，来验证用户提交，包括是否填写了所有必填字段，是否符合其他指定的要求。验证完以后，就可以将提交作

为一个单独的文档，保存在 submissions 集合中。提交的形式类似于下面这样：

```
comment_submission = {
    :form_id => "comments",
    :name => "Mike D.",
    :email => "mike@example.com",
    :comment => "MongoDB is flexible!"
}
```

这里又一次充分利用了文档模型，每个提交都能有自己的键（这里是 :name,:email 和 :comment）。提交只有一个必选键 :form_id。目的是为了快速获取特定表单的所有提交：

```
db["submissions"].find :form_id => "comments"
```

为了提速，还要按照 :form_id 建立索引：

```
db["submissions"].create_index :form_id
```

同样，对于给定的提交，也能通过 :form_id 找到对应的表单文档。

11.3.4 Ruby的对象映射和在Rails中使用MongoDB

有一些库在基本的 Ruby 驱动的基础上为 MongoDB 的文档提供了模型（model）、确认（validation）、关联（association）等内容。最流行的工具要算 MongoMapper（http:mongo-mapper.com/）和 Mongoid（http://mongoid.org/）了。如果习惯了 ActiveRecord 或者 Data-Mapper，那么你除了基本的 Ruby 驱动外要考虑使用这些对象映射工具。

MongoDB 与 Ruby on Rails 配合完美，尤其是用了上面提到的映射工具时。MongoDB 站点上有 MongoDB 与 Rails 集成的最新指南（http://www.mongodb.org/display/DOCS/Rails+-+Getting+Started）。

11.4 实时分析：Python

MongoDB 的 Python 驱动叫做 PyMongo。本节会用 PyMongo 实现对 Web 应用度量的实时跟踪。PyMongo 最新的帮助文档可在 http://api.mongodb.org/python 上找到。

11.4.1 安装PyMongo

Python Package Index（http://pypi.python.org/pypi/pymongo/）中就有 PyMongo，用 easy_ install（http://pypi.python.org/pypi/setuptools）就可以安装：

```
$ easy_install pymongo
Searching for pymongo
```

```
Reading http://pypi.python.org/simple/pymongo/
Reading http://github.com/mongodb/mongo-python-driver
Best match: pymongo 1.6
Downloading ...
Processing pymongo-1.6-py2.6-macosx-10.6-x86_64.egg
Moving ...
Adding pymongo 1.6 to easy-install.pth file

Installed ...
Processing dependencies for pymongo
Finished processing dependencies for pymongo
```

这样就安装了 PyMongo，同时会试着安装可选的 C 扩展。要是 C 扩展安装失败，一切也能照常运行，只不过性能不高。如果发生这种情况，安装时会出现错误提示。

除了用 `easy_install`，还可以通过源代码检查运行 python setup.py install 来安装 PyMongo。

11.4.2 使用PyMongo

`pymongo.connection.Connection` 类与 MongoDB 服务器连接。现在新建一个连接，用属性的方式访问，以得到 analytics 数据库：

```
from pymongo import Connection
db = Connection().analytics
```

PyMongo 的 API 与 MongoDB shell 的 API 非常相似。和 Ruby 驱动一样，PyMongo 也使用下划线命名法。PyMongo 中用字典来表示文档，因此要插入和获取文档 `{"a" : [1, 2, 3]}`，使用如下命令就可以：

```
db.test.insert({"a": [1, 2, 3]})
db.test.find_one()
```

Python 的字典是无序的，所以 PyMongo 提供了 `dict` 的一个有序的子类 `pymongo.son.SON`。大多数需要顺序的情况下，PyMongo 提供的 API 都将这点对用户隐藏。如果遇到了特殊情况，应用可以使用 `SON` 实例替代字典，来确保文档的键的顺序。

11.4.3 用于实时分析的MongoDB

MongoDB 非常适合做实时环境中的跟踪度量工具，原因如下。

- `upsert` 操作（详见第 3 章）能用一条消息插入新跟踪文档或者对已有文档更新计数器。
- `upsert` 发送后不需要等待回应，记得离弦之箭吧？这样应用程序就能避免在每次分析更新时阻塞。完全没有必要等着看操作是否成功，因为分析代码中的错误用户又不会知道。

- 可以用 `$inc`[1] 更新计数器，而不必分别执行查询和更新操作。这样还能避免同时更新的冲突。
- MongoDB 的更新性能很高，因此为分析的每个请求执行一个或多个更新是合理的。

11.4.4 模式

下面的例子会跟踪网站的页面访问，以小时为统计单位。追踪的内容是总的页面访问次数和每个单独 URL 的访问次数。目的是每小时得到一个集合，包含下面这样的文档：

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)","url" : "/foo",
  "views" :5 }
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)","url" : "/bar",
  "views" :5 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)","url" : "/",
  "views" : 12}
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)","url" : "/bar",
  "views" :3 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)","url" : "/foo",
  "views" :10 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)","url" : "/foo",
  "views" :21 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)","url" : "/",
  "views" : 3}
...
```

每个文档代表对某一小时内某个 URL 的访问次数。如果这小时没有访问，就没有相应的文档。要记录整个网站所有页面的访问次数，应使用另外一个集合 `hourly_totals`，形式如下：

```
{ "hour" : "Tue Jun 15 2010 9:00:00 GMT-0400 (EDT)", "views" : 10 }
{ "hour" : "Tue Jun 15 2010 10:00:00 GMT-0400 (EDT)", "views" : 25 }
{ "hour" : "Tue Jun 15 2010 11:00:00 GMT-0400 (EDT)", "views" : 24 }
...
```

因为是总数，这里的不同就是不需要 `"url"` 键了。若在这个小时内没有页面访问，也就没有相应的文档。

11.4.5 处理请求

每当应用接收了请求，就需要相应地更新分析的集合。无论是对于特定 URL 的 `hourly` 集合还是一般的 `hourly_totals` 集合，都要更新计数。下面定义一个函数来处理某个 URL，更新分析数据：

```
from datetime import datetime
```

译注1：`$inc`是原子的。

```
def track(url):
    hour = datetime.utcnow().replace(minute=0, second=0, microsecond=0)
    db.hourly.update({"hour": hour, "url": url},
                     {"$inc": {"views": 1}}, upsert=True)
    db.hourly_totals.update({"hour": hour},
                            {"$inc": {"views": 1}}, upsert=True)
```

还要建立索引以保证高效地执行更新：

```
from pymongo import ASCENDING

db.hourly.create_index([("url", ASCENDING), ("hour", ASCENDING)],
unique=True)
db.hourly_totals.create_index("hour", unique=True)
```

对于 `hourly` 集合，使用 `"url"` 和 `"hour"` 的复合索引，而对于 `hourly_totals`，只对 `"hour"` 进行索引。两个索引都是唯一索引，因为每个统计单位只应该有一个文档。

每次有请求时，就调用一次 `track`，并将请求的 URL 传递过去。它就能执行两次 upsert，要么创建新的统计单位文档，要么更新已有文档的 `"views"` 键。

11.4.6 使用分析数据

现在已经有了跟踪数据，我们需要一种方法来查询数据并利用这些数据。下面输出最近 10 小时总的页面访问量：

```
from pymongo import DESCENDING

for rollup in db.hourly_totals.find().sort("hour",DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

这个查询能利用已经有的 `"hour"` 上的索引。可以对每个 URL 执行类似的操作：

```
for rollup in db.hourly.find({"url": url}).sort("hour",DESCENDING).limit(10):
    pretty_date = rollup["hour"].strftime("%Y/%m/%d %H")
    print "%s - %d" % (pretty_date, rollup["views"])
```

唯一的不同就是，我们增加了一个查询文档，用于选择 `"url"`。这里也用到了已经对 `"url"` 和 `"hour"` 建立的复合索引。

11.4.7 其他因素

还要考虑的是应该定期运行一个清理任务，删掉旧的分析文档。我们只要显示最近 10 小时的数据，没有必要保留一个月的文档。按照下面的做法可以删除超过 24 小

时的文档，通过 `cron` 或类似机制就可以定期执行了：

```
from datetime import timedelta

remove_before = datetime.utcnow() - timedelta(hours=24)

db.hourly.remove({"hour": {"$lt": remove_before}})
db.hourly_totals.remove({"hour": {"$lt": remove_before}})
```

在这个例子中，前一个清除因为没有在 `"hour"` 上定义索引，所以得做表扫描。要想有效地执行这个操作（或者执行其他根据 `"hour"` 查询所有 URL 的操作），则应该考虑在 hourly 集合中在 `"hour"` 上创建一个索引。

这个例子另一个值得注意的方面是，除了页面访问做别的统计分析也很容易，调整时间单位的大小也一样容易（甚至可以同时使用多种时间单位）。只要稍稍调整 `track` 函数，以给定的时间单位用 upsert 记录需要的度量就好了。

附录 A

安装 MongoDB

在大部分平台上安装 MongoDB 都很简单。Linux、Mac OS X、Windows 和 Solaris 都有对应的预编译二进制版本。这也就意味着在大多数平台上，只需要从 http://www.mongodb.org 下载压缩包，解压，执行安装就好了。MongoDB 需要一个数据目录写入数据库文件，和一个端口用来监听连接。这节会介绍在两种不同的系统下的安装过程：一种是 Windows，一种是其他（Linux、Mac OS X、Solaris）。

当提及"安装 MongoDB"，一般具体指的是构建核心的数据库服务器 `mongod`。`mongod` 无处不在，可以作为单个服务器、主从节点、副本集的成员，还可以当做片。通常就是所需要的 MongoDB 进程。第 8 章介绍了一同下载的其他的二进制文件。

A.1 选择版本

MongoDB 的版本号也非常好理解：偶数的版本号是稳定版，奇数的是开发版。例如，1.6 开头的是稳定版，比如 1.6.0、1.6.1 和 1.6.15。以 1.7 开头的则是开发版，1.7.0、1.7.2、1.7.10 都是开发版。现在就以 1.6/1.7 为例来具体讲一下版本的演变过程。

(1) 开发者发布 1.6.0。这是一个大版本更新，会有很多变化。建议生产系统尽快升级到这个版本。

(2) 开发者开始着手开发 1.8 时，发布了 1.7.0。这个新的开发分支和 1.6.0 非常类似，但会加入一些新特性，还可能引入一些 bug。

(3) 开发者继续添加新功能，然后发布 1.7.1、1.7.2 等。

(4) bug 修正和没什么风险的功能则合并到 1.6 的分支上，就有了 1.6.1、1.6.2 等。对于这种调整是非常保守的，只有个别功能会添加到稳定版中，一般仅修

正 bug。

(5) 1.8.0 的所有里程碑都达到后，开发者发布一个版本，比如，1.7.5。

(6) 在详细测试 1.7.5 后，通常需要修复一些 bug。修复之后则发布 1.7.6。

(7) 反复重复第 6 步，直到没有太明显的新 bug，然后 1.7.6（或者最后发布的版本）就会被重命名为 1.8.0。这样，最新的开发版就成了新的稳定版。

(8) 将所有版本号增加 .2，然后从第一步重新再来。

所以，最初版本的开发分支是非常不稳定的（x.y.0、x.y.1、x.y.2），但当分支进入 x.y.5 的时候，就非常接近可用于生产的水平了。在 MongoDB bug 追踪器（http://jira.mongodb.org）上浏览核心服务器的路线图，可以了解生产版本的发布细节。

除非是需要开发版的某些功能，否则在生产环境中应该用稳定版。即便是需要开发版的某些功能，在用之前，最好通过邮件列表或者 IRC 与开发者取得联系，让他知道你要部署开发版到生产环境中，请他给出一些建议，确保数据的安全。（当然，这样总是一个不错的主意。）

如果项目还在开发阶段，用开发版可能会更好。项目上线部署时，稳定版也发布了（MongoDB 保持每隔几个月就发布稳定版的周期），这样就可以用到最新的功能。但是，必须要权衡利弊，因为这么做很可能引入一些 bug，令一些新用户感到困惑，失去信心。

A.2　在Windows下安装

在 Windows 下安装 MongoDB，先要从 MongoDB 下载页（http://www.mongodb.org/display/DOCS/downloads）下载 Windows 的 zip 文件。要遵循前面的意见，选择合适的 MongoDB 版本。Windows 下有 32 位和 64 位的版本可供选择。点击连接，下载 .zip 文件，然后解压缩。

接着，要建立一个目录，用于存放数据库文件。MongoDB 默认使用 C:\data\db 作为数据目录。可以创建这个目录，也可以在系统的任意位置创建其他空目录。前面讲过，如果用的不是 C:\data\db，则需要在启动 MongoDB 时指明数据目录。

有了数据目录之后，打开命令提示（cmd.exe）。进入到 MongoDB 解压的目录，然后执行：

```
$ bin\mongod.exe
```

如果用的不是 C:\data\db 作为数据目录，得用 `--dbpath` 参数在这里指定：

```
$ bin\mongod.exe --dbpath C:\Documents and Settings\Username\My
                          Documents\db
```

更多选项说明参见第 8 章，或者用 `mongod.exe --help` 查看所有选项。

作为服务进行安装

MongoDB 在 Windows 中还可以作为服务进行安装。使用完整的路径来运行，忽略所有空格并使用 `--install` 选项，就可以安装了。例如：

```
$ C:\mongodb-windows-32bit-1.6.0\bin\mongod.exe
      --dbpath "\"C:\Documents and Settings\Username\My Documents\db\""
      --install
```

之后就可以在控制面板中启动或停止服务了。

A.3 在POSIX系统（Linux、Mac OS X和Solaris）下安装

参考 A.1 节的建议，选择 MongoDB 的版本。到 MongoDB 下载页面，选择操作系统对应的版本。



如果在 Mac 中，要检查是 32 位的还是 64 位的。选择不当的版本，Mac 就会不能启动 MongoDB，还会给出很多令人费解的错误信息。点击左上角的苹果，选择“About This Mac”选项查看相应的信息。

必须要先建立数据目录，以供数据库存放文件。默认的数据目录是 /data/db，但用别的目录也是没问题的。如果创建了默认的数据目录，要确保有写权限。创建目录并设置写权限的操作如下：

```
$ mkdir -p /data/db
$ chown -R $USER:$USER /data/db
```

`mkdir -p` 会创建目录和必要的父目录（就是说，/data 不存在时，会先创建 /data，然后创建 /data/db）。`chown` 更改 /data/db 的所有者，以便用户可以对其写入。当然，可以在你自己的主目录中创建目录，MongoDB 用这样的目录不会遇到权限问题。

解压缩从 http://www.mongodb.org 下载的 .tar.gz 文件。

```
$ tar zxf mongodb-linux-i686-1.6.0.tar.gz
$ cd mongodb-linux-i686-1.6.0
```

然后就可以启动数据库了：

```
$ bin/mongod
```

也可以用 `--dbpath` 选项指定别的数据库路径：

```
$ bin/mongod --dbpath ~/db
```

关于常用选项的介绍，详见第 8 章，或者使用 `mongod --help` 查看所有选项。

用包管理器安装

这些系统上有很多包管理器可以安装 MongoDB。比如，Debian 和 Ubuntu 就有官方包，Red Hat、Gentoo 和 FreeBSD 下有非官方包。如果用的是非官方版本，启动数据库时要查看日志，有时候这些包没有 UTF-8 支持。

在 Mac 上也有 Homebrew 和 MacPorts 的非官方包。如果用 MacPorts 版本，一定得小心：编译 MongoDB 需要的 Boost 库可能要花费数小时。下载后，把编译工作留给无尽的长夜吧。

无论使用哪种包管理器，在出问题之前就搞清楚 MongoDB 将日志存放在什么地方，未雨绸缪总没什么坏处。一定要确保日志保存正常，以备不时之需[1]。

译注1：日志监控可以避免大量的血泪教训。

附录 B

mongo：MongoDB shell

整本书中到处都用了 mongo，其实它是数据库 shell。一般假定它和 mongod 运行在同一台机器上，还假定 mongod 绑定了默认端口。如果不是这样的话，可以在启动时指定这些参数，让 shell 连接另一台服务器：

```
$ bin/mongo staging.example.com:20000
```

这样就会连接运行在 staging.example.com 上端口为 20000 的 mongod。

shell 默认连接 test 数据库。要使用别的数据库，在服务器地址后添加斜杠和数据库名就可以了：

```
$ bin/mongo localhost:27017/admin
```

这样连接的就是本地默认端口的 mongod，但用的是 admin 数据库。

也可以用 --nodb 选项启动 shell，而不连接数据库。如果只是试试 JavaScript，过一会儿再连数据库，就可以这么用：

```
$ bin/mongo --nodb
MongoDB shell version: 1.5.3
type "help" for help
>
```

一定要记住，db 绝不是仅有的数据库。从 shell 中可以连接任意多个数据库，这对多个服务器的环境还是非常方便的。调用 connect()，并将结果赋值给变量。例如，在分片环境中，可能想用 mongos 表示 mongos 服务器，还想要有到每个片的连接，可以如下操作：

```
> mongos = connect("localhost:27017")
connecting to: localhost:27017
localhost:27017
> shard0 = connect("localhost:30000")
connecting to: localhost:30000
localhost:30000
> shard1 = connect("localhost:30001")
connecting to: localhost:30001
localhost:30001
```

随后，就能将 `mongos`、`shard0` 和 `shard1` 作为 `db` 变量使用（但一些特殊的辅助方法不好用，比如 `use foo` 或者 `show collections`）。

shell工具

还有些有用的 shell 函数前面没有讲到。

对于管理多个数据库，有多个数据库变量就比简单的 `db` 有用处。例如，在分片中，要维护一个单独的指向配置服务器的变量：

```
> config = db.getSisterDB("config")
config
> config.shards.find()
...
```

用 `connect` 函数可以在一个 shell 中连接多个服务器：

```
> shard_db = connect("shard.example.com:27017/mydb")
connecting to shard.example.com:27017/mydb
mydb
>
```

shell 下还可以运行 shell 命令：

```
> runProgram("echo", "Hello", "world")
shell: started mongo program echo Hello world
0
> sh6487| Hello world
```

（输出看上去有点奇怪，因为 shell 还在运行中。）

附录 C

深入 MongoDB 内部

大多数情况下，MongoDB 的用户可将其视为黑盒。若想理解性能特点或者深入了解系统，就得深入 MongoDB 内部了。

C.1 BSON

MongoDB 的文档是个抽象概念。其具体的呈现形式取决于使用的驱动和编程语言。因为 MongoDB 中的通信大量依赖于文档，所以需要一种所有驱动、工具和进程都能共享的文档表达方式。这种表达叫做 Binary JSON（BSON）。

BSON 是轻量的二进制格式，能将 MongoDB 的所有文档表示为字节字符串。数据库能理解 BSON，存在磁盘上的文档也是这种格式。

当驱动要插入文档，或者将文档作为查询条件，驱动会将文档转换成 BSON，然后再发往服务器。同样，返回到客户端的文档也是 BSON 格式的字符串。驱动需要将这些数据解码，变成本机的文档表示，最后返回给客户端。

用 BSON 格式的 3 个主要目标。

- 效率
  BSON 设计用来更有效地表示数据，占用更少的空间。最差的情况下，BSON 比 JSON 效率略低；最好的情况下（比如存放二进制数据或者大数），BSON 要高效得多。

- 可遍历性
  有些时候 BSON 牺牲了空间效率，换取更容易遍历的格式。例如，在字符串前面加入其长度，而不是在结尾处使用一个终结符。这对 MongoDB 内省文档很有用。

- 性能

  最后 BSON 的编码和解码速度都很快。它用 C 风格的表现方式来表示类型，在大多数编程语言中都非常快。

关于 BSON 的详细说明，参见 http://www.bsonspec.org。

C.2 Mongo传输协议

驱动在 TCP/IP 协议的基础上简单封装了 MongoDB 传输协议，用来与 MongoDB 交互。MongoDB 的 wiki（http:www.mongodb.org/display/DOCS/Mongo+Wire+Protocol）中对该协议有详细的说明，不过基本上就是一个简单封装的 BSON 数据[1]。例如，插入消息会有 20 字节的头部数据（包括告知服务器执行插入操作的代码，以及消息长度）、要插入的集合名、要插入的 BSON 文档列表。

C.3 数据文件

MongoDB 的数据目录（默认是 /data/db/）中，每个数据库都有几个独立的文件。每个数据有一个 .ns 文件和若干数据文件，数据文件以递增的数字结尾。所以，数据库 foo 会被存放在 foo.ns、foo.0、foo.1、foo.2 等文件中。

每个新的以数字结尾的数据文件大小会加倍，直到达到最大值 2 GB。这是为了让小数据库不浪费太多的磁盘空间，同时让大数据使用磁盘上连续的空间。

MongoDB 为了保证性能还会预分配数据文件（可以用 `--noprealloc` 关闭这一功能）。预分配在后台完成，有数据文件被填满时就自动启动。这就意味着 MongoDB 服务器总是试图为每一个数据库保留一个额外的空数据文件，来避免文件分配所产生的阻塞。

C.4 命名空间和数据域

在数据文件内部，每个数据库都是按照命名空间组织的，一种类别的数据与其他类别的分开存放。每个集合的文档都有自己的命名空间，索引也是。命名空间的元数据存放在数据库的 .ns 文件中。

每个命名空间的数据都被分成若干组，放到数据文件的某一区域内，这个区域称为数据域。在图 C-1 中可以看到数据库 foo 有 3 个数据文件，其中第 3 个是预分配的空文件。前两个数据文件被分成几个数据域，属于几个不同的命名空间。

译注1：MongoDB传输协议和HTTP、FTP一样，是一种应用层协议，只不过这种协议目前只用在MongoDB相关应用中。

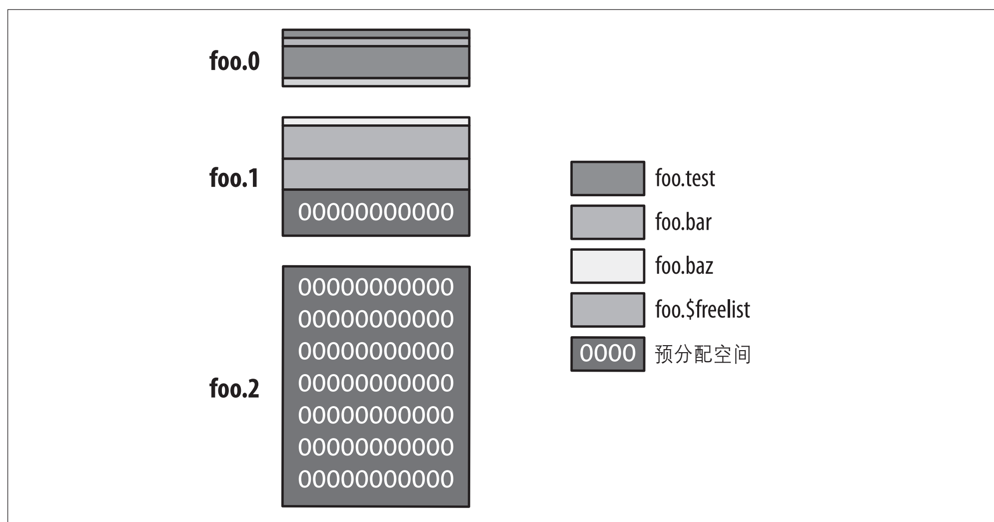


图 C-1：命名空间和数据域

图 C-1 命名空间和数据域还展示了一些关于命名空间和数据域有趣的地方。每个命名空间可以有几个不同的数据域，在磁盘上不（必）连续。类似于数据库的数据文件，每次新分配的命名空间的数据域大小也会增加。这是为了平衡命名空间浪费的空间和尽量让一个命名空间的数据连续做出的折中。图中还有个特殊的命名空间 `$freelist`，存放着不再使用的数据域（例如，删除集合或索引产生的数据域）。当命名空间分配新的数据域时，系统会先查找空闲列表，看看有没有合适大小的数据域可用。

C.5　内存映射存储引擎

MongoDB 默认的存储引擎（也是本书写作时唯一的存储引擎）是内存映射引擎。当服务器启动后，将所有数据文件映射到内存。然后由操作系统来负责将缓冲数据写入磁盘并将数据调入调出内存页面。这样的引擎有若干重要的特性。

- MongoDB 管理内存的代码非常精炼，原因就是将大部分工作推给了操作系统。
- MongoDB 服务器进程的虚拟大小通常会非常大，超过整个数据集的大小。这没关系，因为操作系统会处理让哪些数据常驻内存。
- MongoDB 不能控制数据写入到磁盘的顺序，也就不能用预写日志提供单机的持久性。正在为 MongoDB 开发的一种新的存储引擎会提供这种功能。
- 32 位的 MongoDB 服务器有个限制，每个 `mongod` 最多只能处理 2 GB 数据。这是因为所有数据必须能用 32 位地址访问到。

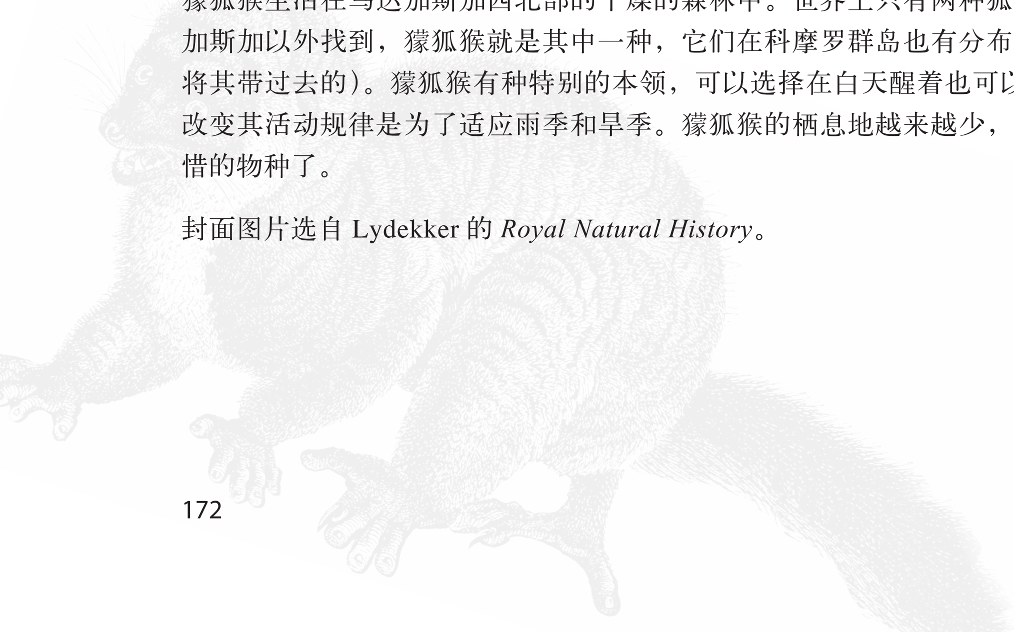
关于封面

本书的封面动物是獴狐猴，是一种马达加斯加特有的灵长类动物。据说狐猴的祖先在6500万年前利用木筏从非洲大陆抵达马达加斯加（行程大约350英里）。狐猴摆脱了与非洲其他物种的竞争（如猴子和松鼠），并且适应了生态圈的各种位置，发展至今种类已接近100种。狐猴（lemur）的得名来自于古罗马神话中的幽灵（lemure），主要因其犹如鬼一般的叫声、昼伏夜出，并有发光的眼睛而得名。马达加斯加的文化中认为狐猴有超自然力，有的将其作为祖先的灵魂，有的作为禁忌之源，还有的作为复仇精神。有的部落将某种狐猴作为本族的祖先。

獴狐猴在狐猴中算是中等体型，大约12～18英寸长，3～4磅重。尾巴有16～25英寸长。雌性和幼年狐猴有白须，雄性有红色须和颊。獴狐猴以水果和花朵为食，是一些植物的传粉者。獴狐猴非常喜欢木棉花的蜜，也吃树叶和昆虫。

獴狐猴生活在马达加斯加西北部的干燥的森林中。世界上只有两种狐猴可以在马达加斯加以外找到，獴狐猴就是其中一种，它们在科摩罗群岛也有分布（据说是人类将其带过去的）。獴狐猴有种特别的本领，可以选择在白天醒着也可以在夜晚醒着，改变其活动规律是为了适应雨季和旱季。獴狐猴的栖息地越来越少，已经是一种珍惜的物种了。

封面图片选自Lydekker的*Royal Natural History*。



MongoDB权威指南

MongoDB如何帮你管理通过Web应用收集的海量数据呢？通过本书的权威解读，你会了解面向文档数据库的诸多优点，会发现MongoDB如此稳定、性能优越甚至能够无限水平扩展背后的原因。

本书的两位作者均来自开发并支持开源数据库MongoDB的公司10gen。数据库开发人员可将此书作为参考指南，系统管理员可以从本书中找到高级配置技巧，其他用户可以了解一些基本概念和用例。你会发现，将数据组织成自包含的JSON风格的文档比组织成关系型数据库中的记录要容易得多。

- 在项目中灵活运用面向文档的存储方式。
- 了解MongoDB的无模式数据模型如何处理文档、集合和多个数据库之间的关系。
- 执行基本的写入操作，构建各种复杂的查询，任何条件下都能查出数据。
- 使用索引、聚合工具，以及其他高级查询技巧。
- 了解监控、安全和认证、备份和修复等内容。
- 建立主从集群和自动故障恢复复制。
- 利用分片水平扩展MongoDB，了解其对应用的影响。
- Java、PHP、Python和Ruby的应用实例应有尽有。

“如同MongoDB本身一样，本书简明扼要、通俗易懂。所有想一探MongoDB究竟的人都需要这本不可或缺的参考手册。”

—— Jeremy Zawodny
*High Performance MySQL*作者

Kristina Chodorow
10gen公司的软件工程师，MongoDB项目的核心成员。负责数据库服务器、PHP驱动、Perl驱动等模块。她曾在世界级的大会上发言，包括OSCON、LinuxCon、FOSDEM和Latinoware。

Michael Dirolf
10gen公司的软件工程师，MongoDB的Python驱动PyMongo的主要负责人，MongoDB Ruby驱动的前任负责人。他曾在世界各地的大型业界会议上发言，推广MongoDB。

适合有一定编程经验的初学者

图灵网站：www.turingbook.com 热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

O'REILLY®
oreilly.com

分类建议 计算机 / 数据库 / MongoDB

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-25112-1
定价：39.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn